Proyecto Fin de Máster
en
Ingeniería de Computadores

"Máster en Investigación en Informática, Facultad de Informática, Universidad Complutense de Madrid"

Energy Impact of Loop Buffer Schemes for Embedded Systems

Autor: Antonio Artés García
Director: José Luis Ayala Rodrigo
Profesor del Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Facultad de Informática
Universidad Complutense de Madrid

Curso academico: 2009-2010

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Energy Impact of Loop Buffer Schemes for Embedded Systems", realizado durante el curso académico 2009-2010 bajo la dirección de José Luis Ayala Rodrigo en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Antonio Artés García

A ti,
suerte de mi vida

Acknowledgements

The author would like to thank those who have had the needed patience to the author after hundreds hours of work. For you, this manuscript.

Abstract

Energy consumption in embedded systems is partially dominated by the consumption of the Instruction Memory Organization. Therefore, any architectural enhancement in this block of the system will cause a reduction in the energy consumption of the total energy budget of the system. Loop buffering is a well known effective scheme to reduce energy consumption in the Instruction Memory Organization.

This manuscript presents a new classification of architectural enhancements and architectures that are based on the use of loop buffer concept. Moreover, an energy design space exploration of different architecture variants, which are based on the classification, is performed. Besides, their energy impacts are analyzed over different real-life embedded application domains widely used in biomedical wireless sensor nodes. The loop buffer organizations, in which this last analysis is focused on, are the single and the banked central loop buffer architectures. The evaluation is performed using TSMC 90nm Low Power library and commercial memories.

Gate-level simulations demonstrate that a trade-off exists between the complexity of the loop buffer architecture and the energy benefits of utilizing it. Besides, if the application has loops with small to medium execution time percentage of the total application execution time, the use of loop buffer architectures in order to bring energy benefits to the system should be very carefully evaluated. From the energy design space exploration, we can see that energy savings from 68% to 74% of the total energy budget of the system can be achieved. Based on the energy analysis performed, it is also demonstrated that energy savings related with the use of multiple or distributed loop buffer architectures are not related with the instruction level parallelism that they introduce. The energy savings are achieved adapting the loop buffers to the loop body sizes of the loops that form the application.

Key words

Energy, consumption, embedded system, Instruction Memory Organization, loop buffer, program memory, instruction cache, design space exploration, biomedical application.

Resumen

El consumo de energía en sistemas empotrados se encuentra fuertemente influenciado por el conjunto de memorias que forman lo que se denomina OMI (Organización de la Memoria de Instrucciones). Por lo tanto, cualquier mejora sobre la arquitectura de este bloque del sistema ocasionará una reducción en el consumo total de energía. Una de las técnicas más conocidas para reducir eficazmente el consumo de energía de esta parte del sistema es el almacenado de bucles (*loop buffering*).

El trabajo contenido en este manuscrito presenta una novedosa clasificación de las arquitecturas y mejoras arquitectónicas que se basan en el uso del concepto de almacenado de bucles. Por otra parte, se realiza una exploración desde el punto de vista energético del espacio de diseño de estas arquitecturas, con el objetivo de analizar las repercusiones energéticas en diferentes dominios de aplicaciones empotradas. Estos dominios de aplicaciones encuentran su utilidad en el campo de las redes inalámbricas de sensores biomédicos. Las arquitecturas sobre las que se centra este último análisis, son la arquitectura de *loop buffer* central y la arquitectura de *loop buffer* implementada en bancos de memoria. La evaluación energética de estas arquitecturas se ha realizado con librerías TSMC 90nm de baja potencia, y memorias comerciales.

La existencia de un compromiso entre la complejidad de la arquitectura *loop buffer* y los beneficios energéticos de utilizarla queda demostrada mediante simulaciones a nivel de puertas. Además, si la aplicación posee bucles cuyo porcentaje de tiempo de ejecución va de pequeño a medio valor, el uso de este tipo de arquitecturas con el fin de aportar beneficios energéticos debe ser cuidadosamente evaluado. De la exploración energética del diseño, podemos ver que el ahorro de energía que se puede lograr es un $68\% - 74\%$ del total de energía consumida por el sistema. En base a este análisis realizado, también se demuestra que el ahorro de energía alcanzado con el uso de arquitecturas múltiples o distribuidas de *loop buffer* no está relacionado con el paralelismo a nivel de instrucción que estas arquitecturas introducen. El ahorro de energía se logra al adaptar la arquitectura al tamaño de los cuerpos de los bucles que forman la aplicación.

Palabras clave

Energía, consumo, sistema empotrado, *loop buffer*, memoria de programa, cache de instrucciones, exploración del espacio de diseño, aplicación biomédica.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this Chapter, a global view of the energy consumption problem in embedded systems is presented in Section 1.1. The approach to solve this problem, which is the core of the work contained in this document, is described step by step in Section 1.2, whereas Section 1.3 presents the motivation to choose the selected approach.

1.1 Background

Embedded systems have different characteristics compared to general-purpose systems. On one hand, they combine software and hardware to run a fixed and specific set of applications, that range from multimedia consumer devices to industrial control systems. These sets of applications differ greatly in their characteristics. They demand different hardware architectures to maximize performance and minimize cost, or make a trade-off between performance and cost according to expected objectives. On the other hand, unlike general-purpose systems, embedded systems are characterized by restrictive resources and low energy budget. In addition to these restrictions, embedded systems have to provide high computation capability and meet real-time constraints. Embedded systems work as reactive systems because they are connected to the physical world through sensors and they have to react to external *stimuli*. Therefore, they have to satisfy varied, tight and time conflicting constraints in order to make themselves reliable and predictable.

All these diverse constraints on embedded systems (i.l., production cost, area, performance and power consumption), result in a NP-complete problem during the design process. Due to this complexity, we have to choose the correct design style which includes a subset of the diverse constraints mentioned previously. Figure 1.1 from Reference [4] shows the main design choices. ASICs (Application-Specific Integrated Circuits) are known for being the best design in terms of performance and energy efficiency, whereas DSP (Digital Signal Processing) processors offer flexibility as well as deliver performance, but they are not energy efficient. However, ASIPs (Application Specific Instruction-set Processors) try to reach the characteristics of an ASIC while still being flexible. Because embedded systems require energy efficiency and certain flexibility, the work presented in this document will be focused on ASIPs.

As embedded system designers, one of our goals is to bring the ASIP based programmable

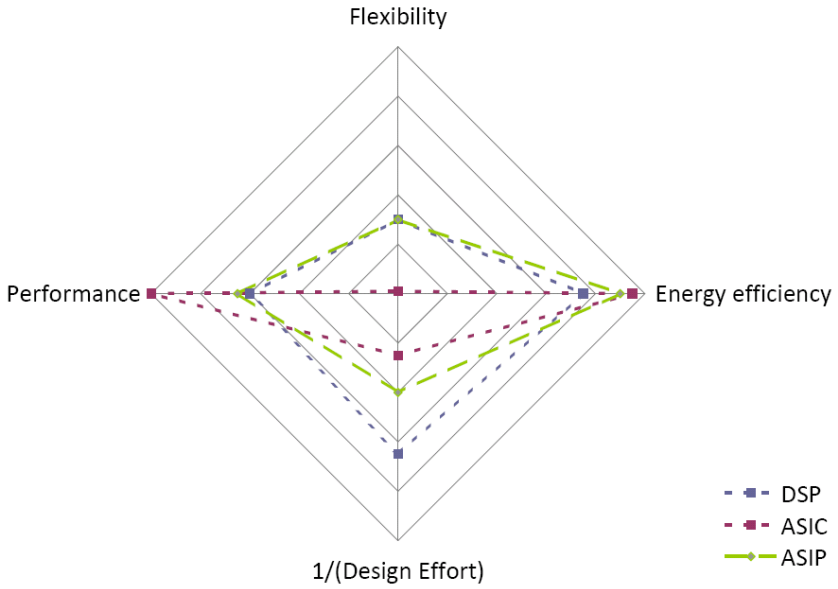


Figure 1.1: Different design styles target different design metrics. Reference [4].

solution as close as possible to the ASIC based solution in terms of energy efficiency. We want to meet the needed performance and flexibility at a minimum price (energy per area unit), rather than achieve higher performance at a higher price. In others words, we pursuit to reduce the energy per task while we provide the required real-time constraints. We remark that to reduce power consumption is not the same as to reduce energy per task, because the latter also takes into account the application execution time. Therefore, in order to optimize an ASIP processor architecture, the designer must evaluate the requirements of the application, versus the performance and energy consumption of the system.

The most important parts of an embedded system are the processor, the Data Memory Hierarchy, the Instruction Memory Organization and the communication network. During the last years, research trends have headed for Data Memory Hierarchy and communication network improvements resulting on a large body of work on these subjects. However, we see a lack of research in the field of the Instruction Memory Organization.

The memory bottleneck in a modern computer system is a widely known problem: the memory speed cannot keep up with the processor speed. Over the past 30 years, the speed of computer systems grew at a phenomenal rate of $50 - 100\%$ per year, whereas during the same period, the speed of typical DRAMs (Dynamic Random Access Memories) grew at a modest rate of about 7% per year. Nowadays, the extremely fast microprocessors spend a large number of cycles idle, waiting for the requested data to arrive from the slow memories. This fact leads to the problem, also known as the *memory wall problem*, in which the performance of the entire system is not governed by the speed of the processor but by the speed of the memory. Figure 1.2 plots processor performance projections against the historical performance improvement in time to access the main memory. See References [10] and [22] for details.

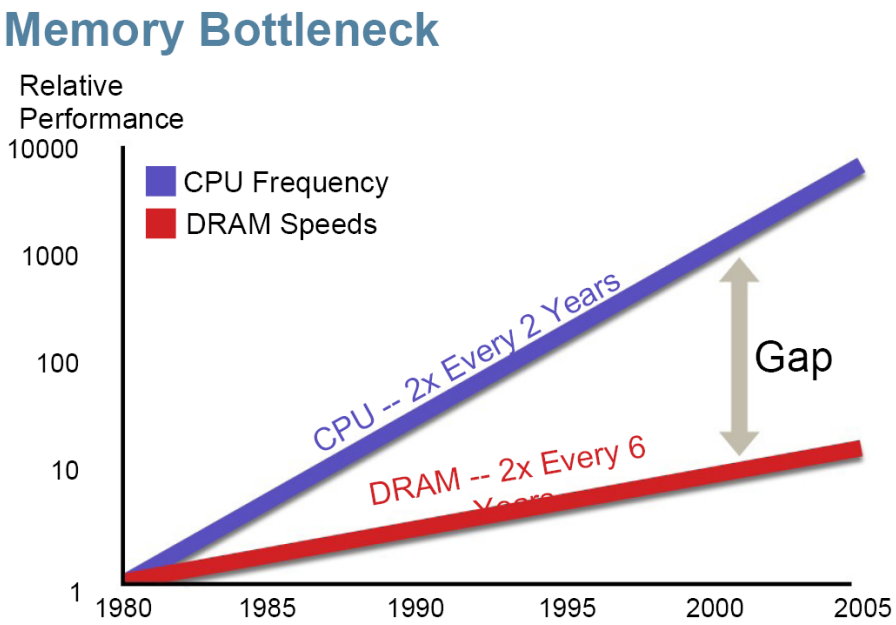


Figure 1.2: Processor performance projections against the historical performance improvement in time to access main memory. Picture from `www.sun.com`.

This well known problem of the memory wall in computer systems becomes even worse in embedded systems, where designers not only need to consider the performance, but also the energy consumption. In an embedded system, memory hierarchies take portions of chip area and power consumption which are not negligible. Several works like [4], [10] and [22] have demonstrated that the memory hierarchy subsystems now account for $50-70\%$ to the total power budget of the instruction-set processor platform. In Reference [22], extensive experiments with an ARM processor based setup are performed in order to validate the above observations. From the results of these experiments, we can see that the memory subsystems consume 65.2% of the total energy budget.

Reference [4] presents another example with different kinds of embedded systems. Figure 1.3 shows a power breakdown for this embedded platform, with the components of the processor core grouped. Indeed, in this Figure, we can see the relative power consumption values per each one of the basics components of an embedded system.

From Figure 1.3, we can see that the power consumption of the Data Memory Hierarchy and the Instruction Memory Organization (Loop Buffer and program memory) represent approximately two thirds of the pie-chart. With this result, we can see that optimizing the Instruction Memory Organization to reduce energy consumption while meeting the required performance becomes extremely important. However, we have to clarify that Data Memory Hierarchy and Instruction Memory Organization are completely different architectures. Depending of the characteristics of the application we are running, the performance or/and the energy consumption behaviors can be opposite. Due to the differences of Data Memory Hierarchy and Instruction Memory Organization, the optimizations related to them are different.

Figure 1.4 shows a typical embedded system architecture consisting of a processor core, a

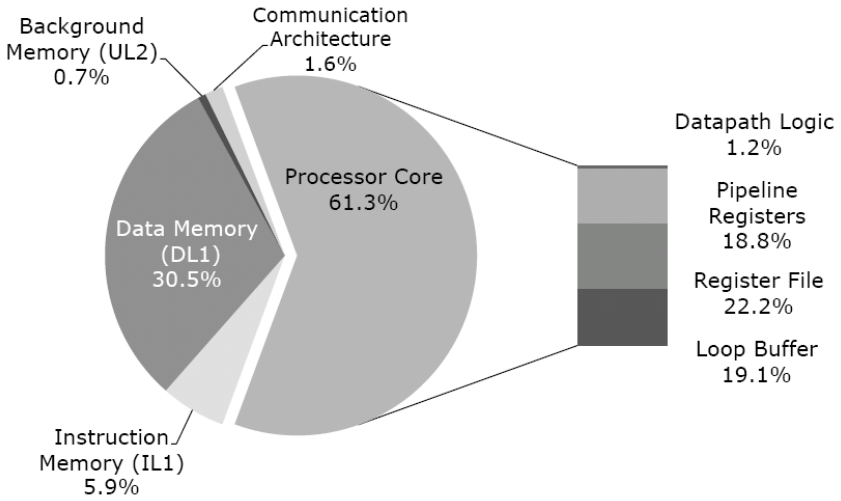


Figure 1.3: Power breakdown for an embedded platform, running a video encoder/decoder application. Reference [4].

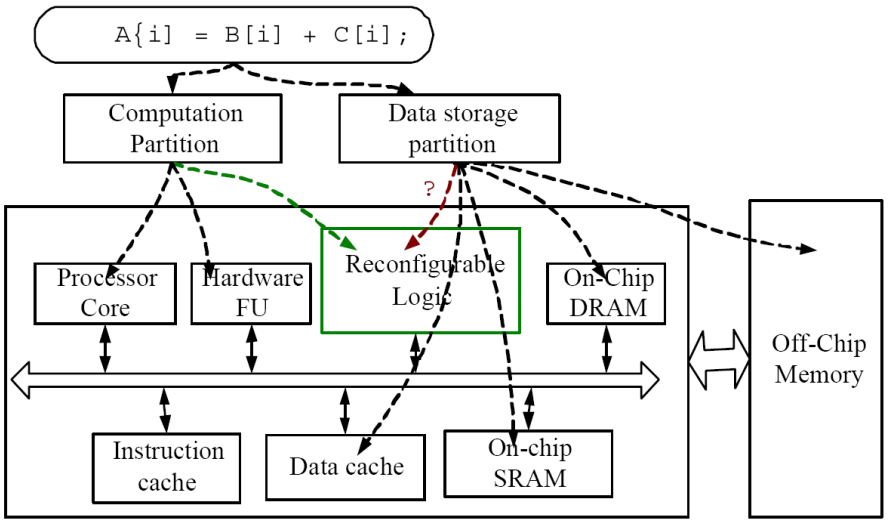


Figure 1.4: A typical embedded system architecture. Reference [7].

reconfigurable hardware, an instruction cache, a data cache, an on-chip scratch memory, an on-chip DRAM, and an off-chip memory. As shown in the Figure 1.4, the computations are partitioned into different computational units while the data is assigned to different storage components. Unlike the memory hierarchies of a general-purpose system mainly concerned with performance, the memory hierarchies of an embedded system have more diverse design objectives, including area, performance, and energy consumption.

1.2 Objectives

The main goal of this master thesis is to analyze and apply one of the most efficient architectural enhancement to reduce energy consumption in embedded systems: the loop buffer concept. From our knowledge, there is not work that had evaluated this architectural enhancement using post-layout simulations to have an accurate estimation of parasitics and switching activity.

Several tasks have been done in order to achieve this goal:

1. Study of related work presenting a novel architectural classification based on three scenarios where all loop buffer based architectures can fall in. This classification is as follows:

   (a) Central loop buffer architectures for single processor organization.

   (b) Multiple loop buffers architectures with shared loop-nest organization.

   (c) Distributed loop buffer architectures with incompatible loop-nest organization.

2. Design and implementation of an experimental framework.

   The experimental framework is used by the energy design space exploration of the loop buffer concept. It contains an ASIP, a Data Memory Hierarchy, an Instruction Memory Organization and an IO interface. The loop buffer concept is implemented in the Instruction Memory Organization.

3. Analysis of the energy design space exploration of the loop buffer concept.

   Using the experimental framework, the energy design space exploration shows the energy trends of these architectures, and it supplies some insights in order to built efficient architectures from the point of view of the energy consumption.

4. Design, implementation and analysis of the loop buffer concept over two real-life embedded applications.

   Based on the knowledge obtained from the energy design space exploration, Instruction Memory Organizations are implemented for specific applications. An analysis of the results from the implementation of the loop buffer concept over these specific applications is performed.

5. Analysis of the results.

A summary of the conclusions from the energy design space exploration and the case studies is presented.

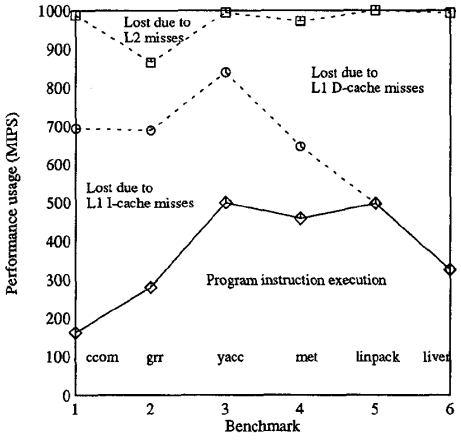
In the following Chapters all these items are described.

1.3 Motivation

In the previous Section 1.1, we have presented the problem related with the memory hierarchies that embedded systems have. In order to solve this problem, embedded system designers have tried to close the processor-memory gap as well as to minimize the energy consumption of the memory hierarchies. However, it has to be well understood that a perfect solution for both problems does not exit.

Our work is only focus on the reduction of the energy per task, taking always into account the required constraints that embedded systems have. On one hand, these systems have either a battery as an energy supply or a scavenge system in order to get the required energy from the environment. On the other hand, as we mentioned at the beginning of Section 1.1, one of our targets is also to meet the performance needed at a minimum price, rather than achieve higher performance at a higher price. The price in our design is energy consumption per task.

In Reference [13], we can see the importance of the use of memory hierarchies. Figure 1.5 provides the results of the evaluation of a system that uses a memory hierarchy based on a normal cache. In the picture, the region below the solid line gives the net performance of the system, while the region above the solid line gives the performance lost in the memory hierarchies.



program name	dynamic instr.	data refs.	total refs.	program type
ccom	31.5M	14.0M	45.5M	C compiler
grr	134.2M	59.2M	193.4M	PC board CAD
yacc	51.0M	16.7M	67.7M	Unix utility
met	99.4M	50.3M	149.7M	PC board CAD
linpack	144.8M	40.7M	185.5M	100x100 numeric
liver	23.6M	7.4M	31.0M	LFK (numeric)
total	484.5M	188.3M	672.8M	

Figure 1.5: Baseline design performance and test program characteristics. Reference [13].

Analyzing Figure 1.5, we can see that the greatest leverage on system performance is obtained by improving the memory hierarchy performance, and not by attempting to further increase the performance of the CPU. Figure 1.5 is also a good example to demonstrate that the impact in performance of the Data Memory Hierarchy and the Instruction Memory Organization depends on the application running in our embedded system. From energy consumption point of view, the same scenario appears.

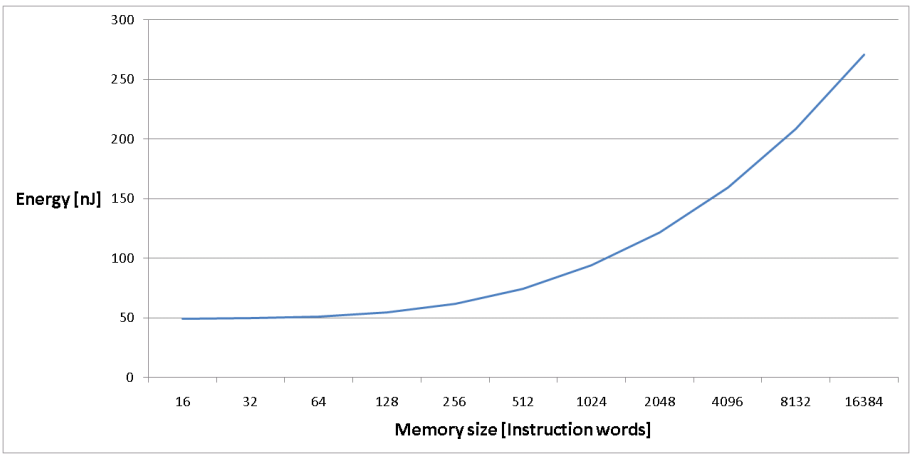


Figure 1.6: Energy consumption per access in SRAM memories designed by Virage Logic Corporation tools [5].

Due to the nature of the heterogeneous architecture, as well as the tightly-coupled hardware and software of embedded systems, many research issues appear involving both architecture and software optimizations. During the last years, several techniques to improve at low cost the characteristics of the baseline Instruction Memory Organization have been performed. In order to make easy the analysis, we can categorize them into two approaches. The first approach deals with the architectural aspect, where designers customize the memory hierarchy by analyzing specific applications, including the parameterization of the data cache size and line size, instruction cache size, scratch memory size, etc. On the other hand, the second approach deals with the software aspect, where designers analyze and optimize the application intensively, such as partitioning data into different types of storage, optimizing the data layout to reduce the amount of cache misses, etc.

Loop buffering is an effective hardware scheme to reduce energy consumption in the Instruction Memory Organization. In signal and image processing applications, a significant amount of execution time is spent in small program segments. Reference [23] presents a study on the loop behavior of embedded applications which demonstrates that 77% of the execution time of an application is spent in loops with 32 instructions or less. With loop buffering, it is possible to store these small program segments in smaller memory banks (e.g. in the form of loop buffers), which have less energy per access, reducing the total energy consumption of the instruction fetch stage significantly. This is due to the fact that accesses to memories of smaller sizes have smaller energy dissipation as shown Figure 1.6.

In this document, a design space exploration of the loop buffer concept from energy consumption point of view is presented. Our analysis introduces a novel architectural classification based on three scenarios where all loop based architectures can fall in. The characterization of the mentioned architectures, the demonstration of this classification into the existing state-of-the-art, and the study of the energy impact of each scenario are performed along this document. Moreover, real-life embedded applications mapped on nodes of biomedical wireless sensor networks are used as case studies to present the energy reduction achieved using Instruction Memory Organizations based on the loop

buffer concept. The Instruction Memory Organizations are implemented based on the analysis of the design space exploration performed previously. To evaluate the energy impact, a post-layout simulation is used to have an accurate estimation of parasitics and switching activity. The evaluation is performed using TSMC 90nm Low Power library and commercial memories.

Chapter 2

Related work

During the last 10 years, researchers have demonstrated that the energy consumption in the Instruction Memory Organization is not negligible. Reference [4] proves, based on a case study, that the Instruction Memory Organization can contribute to a large percentage (40%) of the total energy consumption of the system. Enhancements to reduce this energy consumption make use of loop buffers. In order to study the energy efficiency of the loop buffer concept in several architectures, a classification is presented where all loop buffer based instruction memory organizations can be grouped in.

The first classification is the most traditional usage of the loop buffer concept. It groups all loop buffer based Instruction Memory Organizations with **central loop buffer architectures for single processor organization**. References [1], [2], [11], [13], [14], [19], [24] and [29] are examples of the work done in this set of architectures.

N. P. Jouppi [13] analyzes three hardware techniques to improve direct-mapped cache performance: *miss caching*, *victim caching* and *stream buffers prefetch*. Chuanjun Zhang [29] proposes a configurable instruction cache, which can be tuned in order to utilize the sets efficiently for a particular application, without any increase in the cache size, associativity, or cache access time. Koji Inoue et al. [11] propose an alternative approach to detect and remove unnecessary tag-checks at run-time. Using execution footprints that are recorded previously in a branch target buffer, it is possible to omit the tag-checks for all instructions in a fetched block. If loops can be identified, fetched and decoded only once, Raminder S. Bajwa et al. [1] propose an architectural enhancement that can switch off the fetch and decode logic. The instructions of the loop are decoded and stored locally, from where they are executed. The energy savings come from the reduction in memory accesses as well as the lesser use of the decode logic. In order to avoid any performance degradation, Lea Hwang Lee et al. [15] implement a small instruction buffer based on the definition, detection and utilization of special branch instructions. This architectural enhancement has neither an address tag store nor valid bit associated with each loop cache entry. Johnson Kin et al. [14] evaluate the *Filter Cache*. This enhancement is an unusually small first-level cache that sacrifices a portion of performance in order to save energy. The program memory is only required when a miss occurs in the *Filter Cache*, otherwise it remains in standby mode. Based on this special loop buffer, K. Vivekanandarajah et al. [24] present an architectural enhancement that detects the opportunity to use the *Filter Cache*, and enables or disables it dynamically. Also, Weiyu Tang et al. [19] introduce a

Decoder Filter Cache in the Instruction Memory Organization to provide directly decoded instructions to the processor, reducing the use of the instruction fetch and decode logic. On the other hand, Nikolaos Bellas et al. [2] propose a scheme, where the compiler generates code in order to reduce the possibility of a miss in the loop buffer cache. The drawback of this work is the trade-off between the performance degradation and the power savings, which is created by the selection of the basic blocks.

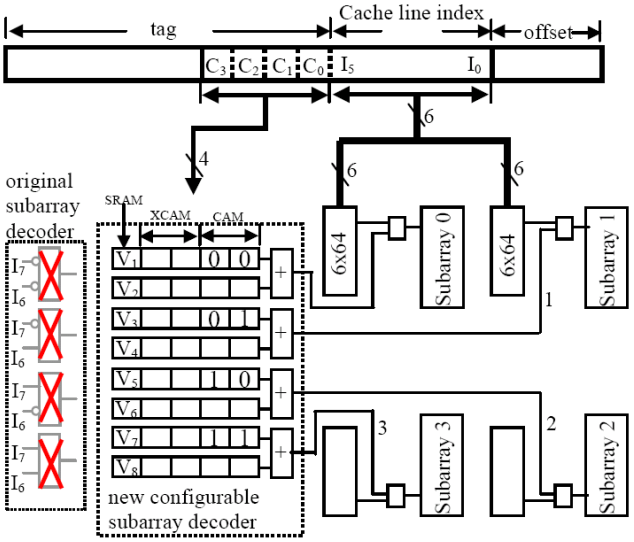


Figure 2.1: The organization of an *efficient cache*. Reference [29].

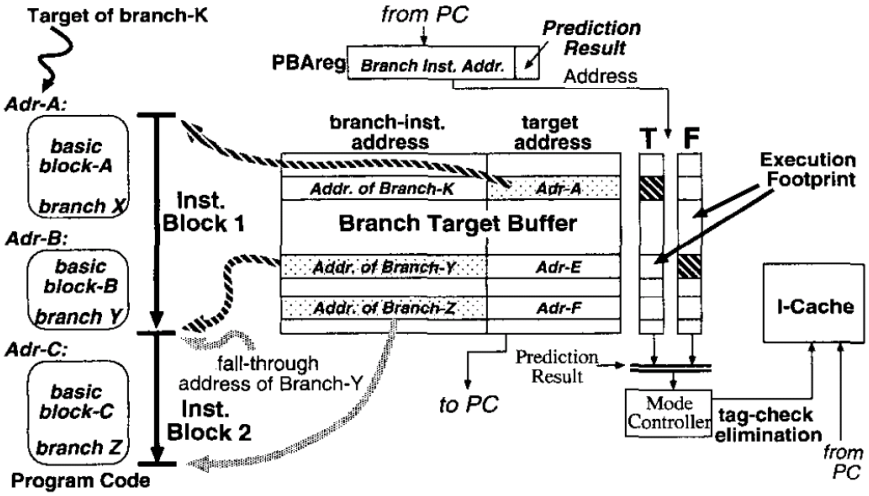


Figure 2.2: The HBTC implementation scheme. Reference [11].

Parallelism is a well known solution in order to increase performance efficiency. Due to the fact that loops form the most important part of an application, loop transformation techniques are applied to exploit parallelism within loops on single-threaded architectures. Centralized resources and global communication make these architectures less energy

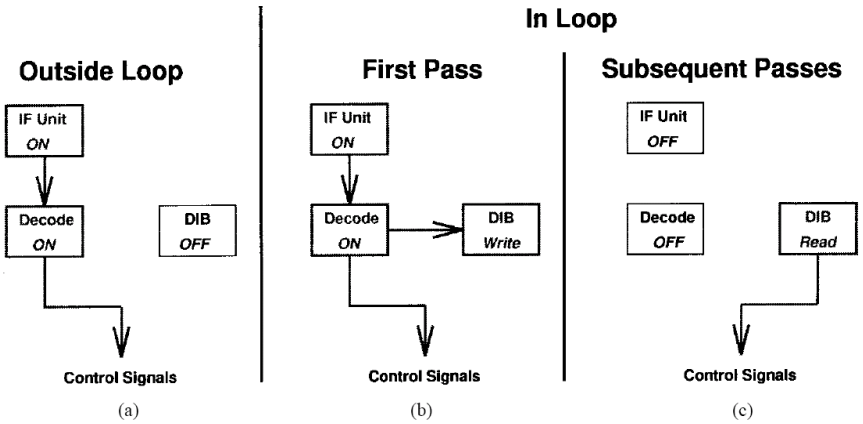


Figure 2.3: DIB configuration in (a) non-loop state, (b) first-pass state, and (c) power-save state. Reference [1].

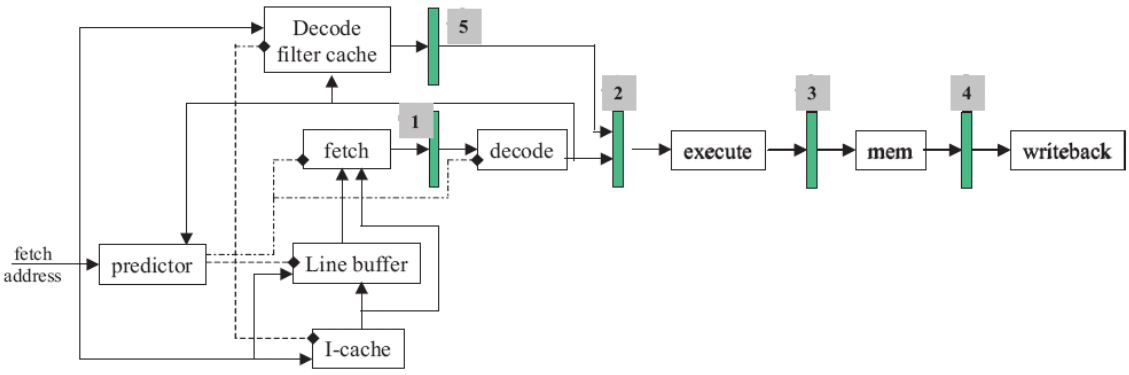


Figure 2.4: Pipeline architecture in a *Decoder Filter Cache*. Reference [19].

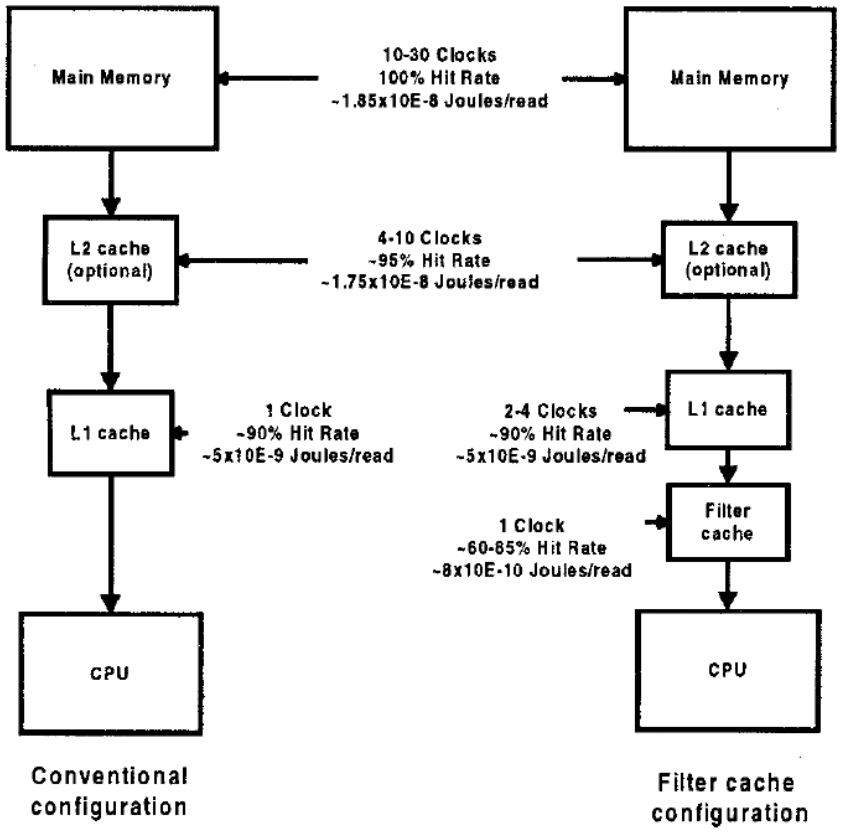


Figure 2.5: Power and performance characteristics of traditional caches and the Filter cache. Reference [14].

efficient. In order to reduce these bottlenecks, several solutions that use multiple loop buffers have been proposed in literature. In our classification, these architectures are classified as **multiple loop buffer architectures with shared loop-nest organization**. References [3], [30] and [31] are examples of the work done in this set of architectures.

Hongtao et al. [30] present a distributed control-path architecture for DVLIW (Distributed Very Long Instruction Word) processors, that overcomes the scalability problem of VLIW control-paths. The main idea is to distribute the fetch and decode logic in the same way that the register file is distributed in a multi-cluster data-path. On the other hand, Hongtao et al. [31] propose a multi-core architecture that extends traditional multi-core systems in two ways. First, it provides a dual-mode scalar operand network to enable efficient inter-core communication without using the memory. Second, it can organize the cores for execution in either coupled or decoupled mode through the compiler. In coupled mode, the cores execute multiple instructions streams in lock-step to collectively work as a wide-issue VLIW. In decoupled mode, the cores execute independently a set of fine-grain communicating threads extracted by the compiler. These two modes create a trade-off between communication latency and flexibility, that it will be optimum depending on the parallelism that we want to exploit. David Black-Schaffer et al. [3] analyze a set of architectures for efficient delivery of VLIW instructions. A baseline cache implementation is compared to a variety of organizations, where the evaluation includes the cost of the memory accesses and the wires which are necessary to distribute the instruction bits.

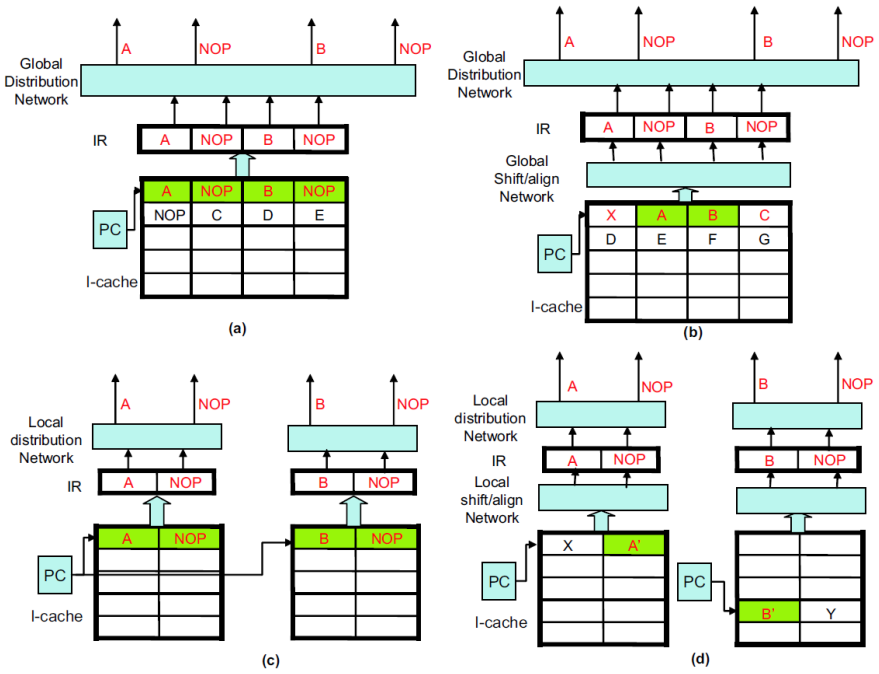


Figure 2.6: VLIW organizations: (a) Centralized instruction cache, uncompressed encoding; (b) Centralized instruction cache, compressed encoding; (c) Distributed instruction cache, centralized PC, uncompressed encoding; (d) Distributed instruction cache, distributed PC, compressed encoding. Reference [30].

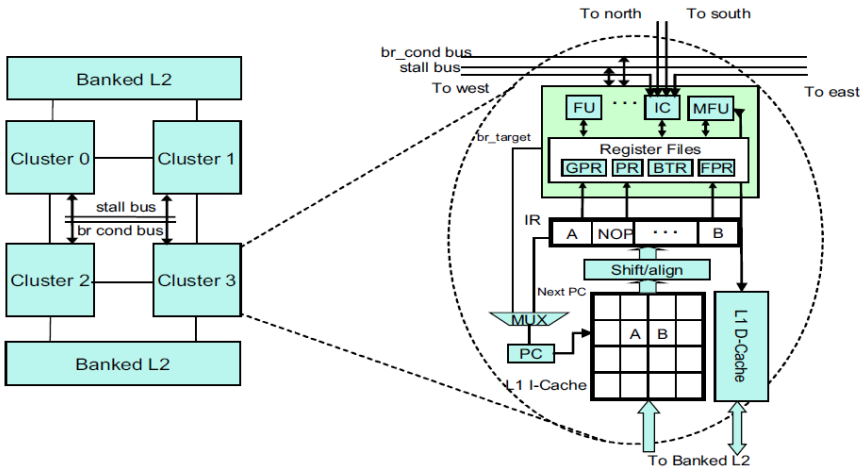


Figure 2.7: DVLIW architecture overview. Four cluster example is shown. Reference [30].

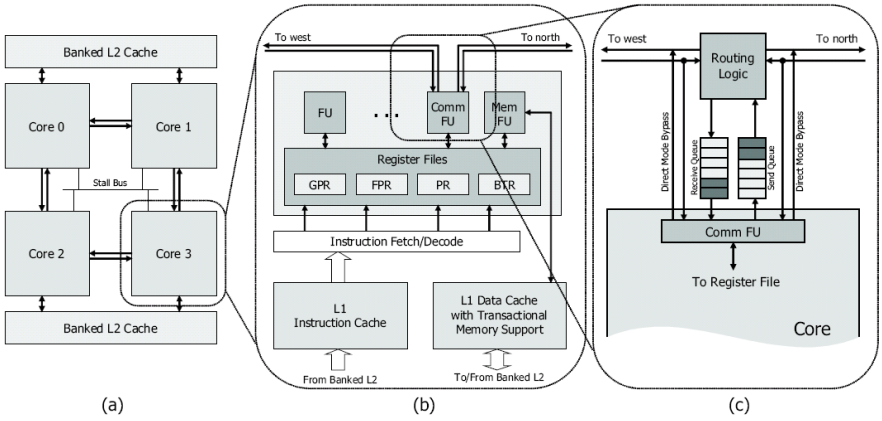


Figure 2.8: Block diagram of the Voltron architecture: (a) 4-core system connected in a mesh topology, (b) Data-path for a single core, and (c) Details of the inter-core communication unit. Reference [31].

The previous architectures have a drawback related with the parallelism efficiency. With these architectures, loops with different threads of control are merged (e.g. using loop fusion) into a single loop with single thread of control. However, incompatible loops cannot be handle by them, because these loops need multiple loop controllers. Hence, not all loops can be efficiently exploited, resulting in loss of performance. A new set of architectures based on distribute loop controllers solves this problem. In our classification, these architectures are named as **distributed loop buffer architectures with incompatible loop-nest organization**. References [9], [12] and [17] are examples of the work done in this set of architectures.

Jayapala et al. [12] propose a low energy clustered Instruction Memory Organization for long instruction word processors. The limit of the benefit of this architecture is performed by a simple profile based algorithm to optimally synthesize the clusters for a given application. Praveen Raghavan et al. [17] present in a multi-thread distributed Instruction Memory Organization that can support parallel execution of multiple incompatible loops. In the proposed architecture, each loop buffer has its own local controller, which is responsible for indexing and regulating accesses to its loop buffer. J.I Gomez et al. [9] present a new loop technique that optimizes the memory bandwidth based on combining loops with an unconformable header. With this technique, the compiler can better exploit the available bandwidth and increase the performance of the system.

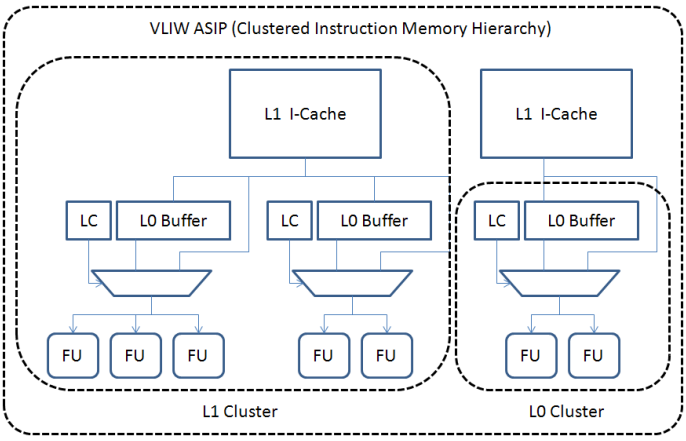


Figure 2.9: Clustered instruction memory hierarchy. Reference [12].

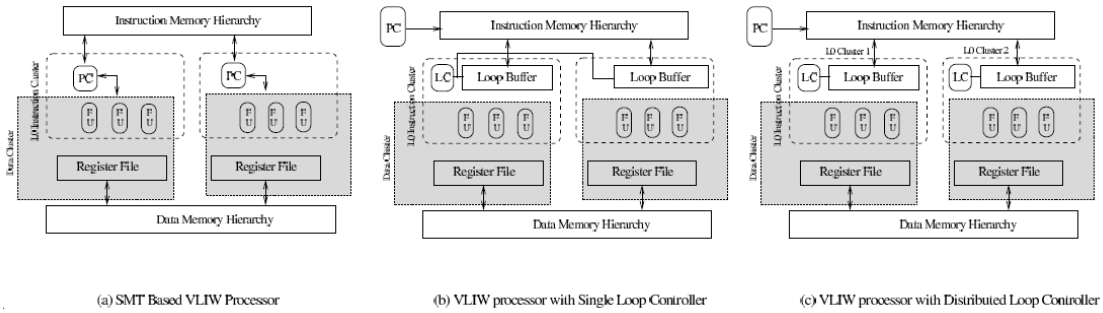


Figure 2.10: Different Processor Architectures supporting Multi-threading. Reference [17].

The presented classification facilities the experimental framework supporting a complete design space exploration of loop buffer concept from energy consumption point of view. Therefore, summarizing, there are 3 sets of architectural enhancements:

- Central loop buffer architectures for single processor organization.
- Multiple loop buffers architectures with shared loop-nest organization.
- Distributed loop buffer architectures with incompatible loop-nest organization.

In Section 3.4, this energy design space exploration is performed showing the energy trends of the architectures presented in this classification.

Chapter 3

Experimental work

In the next Sections, the experimental work is presented. Section 3.1 introduces the processor architecture. Section 3.2 explains the modifications and enhancements done in this processor architecture in order to build the desired experimental framework. Section 3.3 describes the simulation methodology followed in the energy design space exploration, which is presented in Section 3.4, and in the embedded real-life applications used as case studies in Section 3.5.

3.1 Processor Architecture

The framework is made up of a processor with its Data Memory Hierarchy and Instruction Memory Organization. The processor is provided by Target Compiler Technologies [20], and both memory hierarchies are designed with Virage Logic Corporation tools [5]. On one hand, Data Memory Hierarchy is based on a memory, which capacity is 4k words of 16 bits, and its interconnections. On the other hand, the Instruction Memory Organization is based on a memory, which capacity is 2k words of 16 bits, and its interconnections.

The processor architecture is a general-purpose processor that has the following characteristics:

- 16-bit integer arithmetic, bitwise logical, compare and shift instructions. These instructions are executed on a 16-bit ALU and operate on an 8 field register file.
- Integer multiplications with 16-bit operands and 32-bit results.
- Load and store instructions from and to a 16-bit data memory with an address space of 64k words, using indirect addressing.
- Various control instructions such as jumps and subroutine calls and returns.
- Support for interrupts and on chip debugging.

The processor also supports zero-overhead looping control hardware. This feature allows fast looping over a block of instructions. Therefore, once the loop is set using a special instruction, there is no need for additional instructions to control it, and the loop is

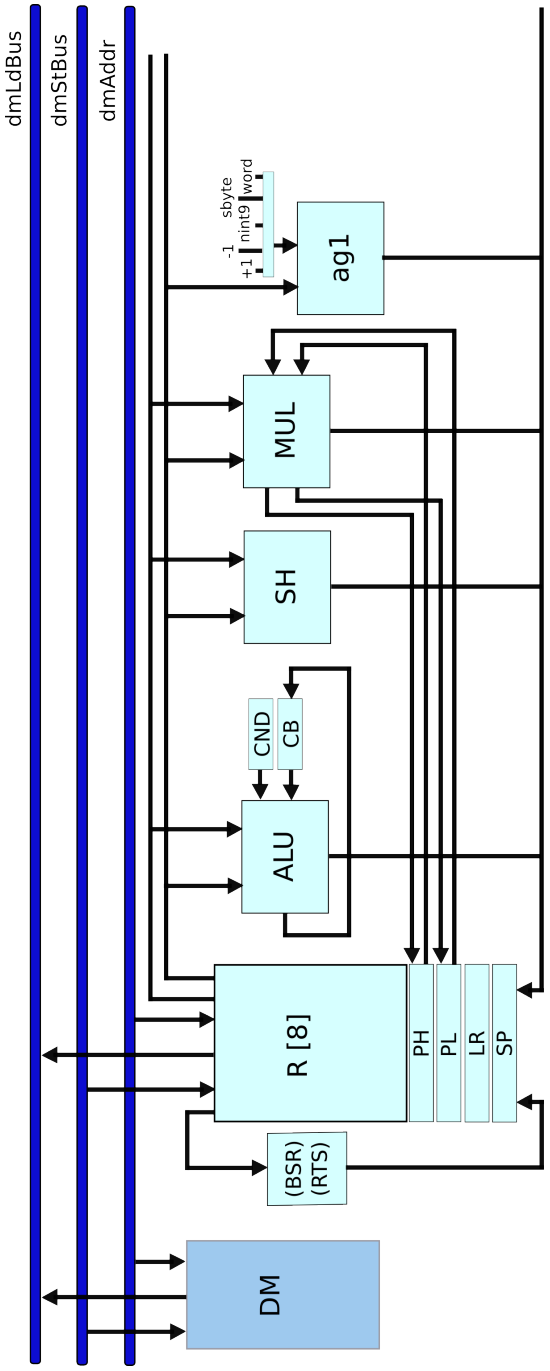


Figure 3.1: Data-path of the general-purpose processor.

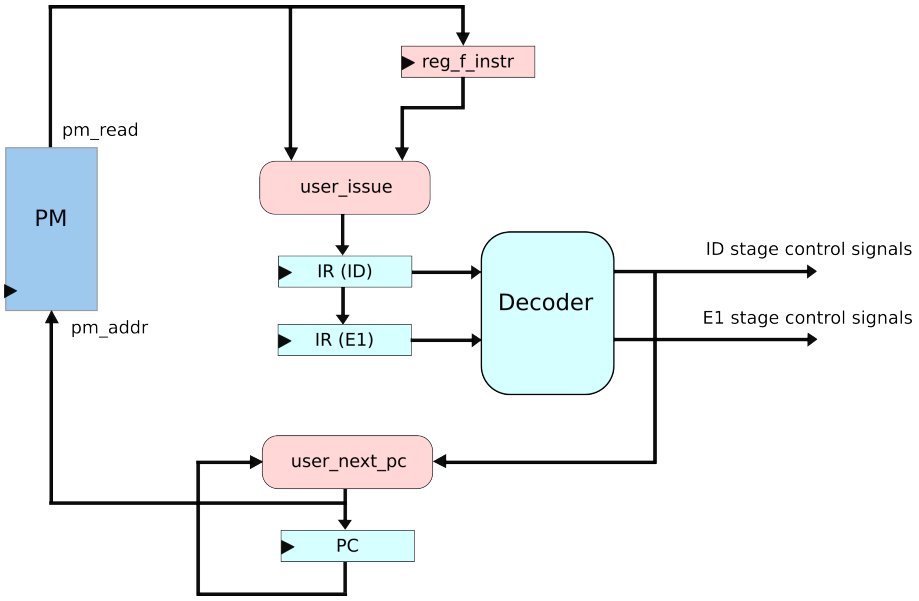


Figure 3.2: Control-path of the general-purpose processor.

executed a pre-specified number of iterations (known at compile time). The status of this dedicated hardware is stored in the following set of special registers:

LS Loop Start address register - It stores the address of the first loop instruction.

LE Loop End address register - It stores the address of the last loop instruction.

LC Loop Count register - It stores the remaining number of iterations of the loop.

LF Loop Flag register - It keeps track of the hardware loop activity.

The special instruction used to control loops takes the values of LC and LE as input parameters. This instruction introduces only one delay slot.

Figure 3.1 presents the data-path of this processor, whereas Figure 3.2 shows the control-path. In Figure 3.1, the main blocks are DM (Data Memory), R (Register File), ALU (Arithmetic Logic Unit), SH (Shift Unit), MUL (Multiplication Unit) and ag1 (address generation unit). In Figure 3.2, the main blocks are PM (Program Memory), PC (Program Counter), and the registers IR(ID) and IR(E1) which are related with the decode and execute stage of the processor pipeline.

3.2 Experimental Setup

To set up the experimental framework, an IO interface is needed. The IO interface used by the experimental framework, in order to provide the capability of receiving and sending data in real-time, is implemented directly in the processor architecture. It uses 16-bit FIFOs as input or output data. They are directly connected to the register file, and new instructions are added to the ISA (Instruction Set Architecture) in order to control them. In this experimental framework text files are used as input and output data.

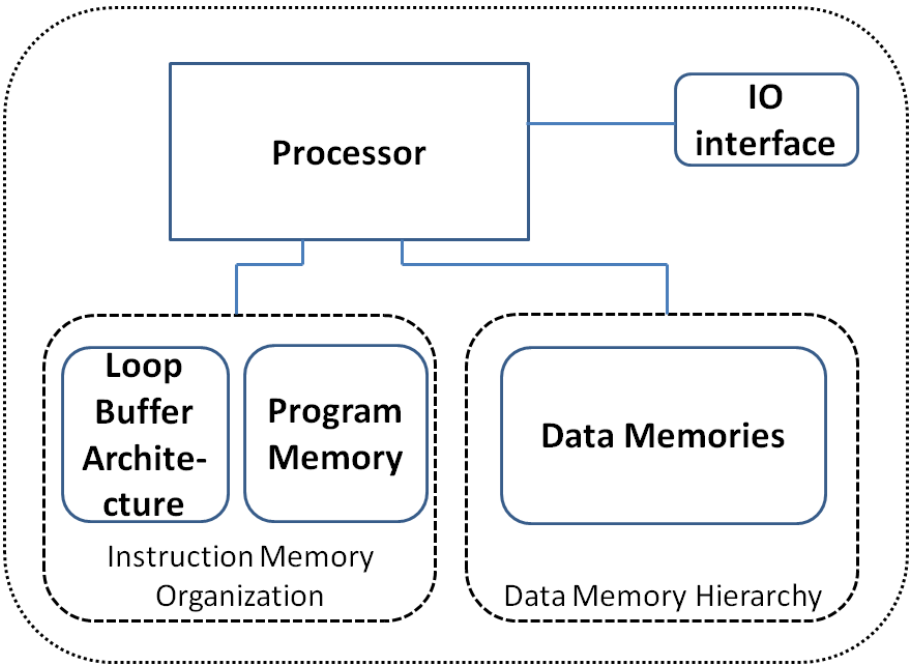


Figure 3.3: Experimental Setup.

Using the system architecture presented in Section 3.1, the loop buffer concept is included in the Instruction Memory Organization. Figure 3.3 depicts the new system architecture including the IO interface and the central loop buffer. Although Figure 3.3 shows a single central loop buffer, our simulation platform is generic enough and can be easily extended to multiple decentralized loop buffer organizations.

In essence, the loop buffer concept operation is as follows. During the first iteration, the instructions are fetched from the program memory to the loop buffer and the processor. The register *LF* changes its value in the first instruction of the loop body. This change is detected by the state-machine in order to set the proper connections between the different components of the Instruction Memory Organization. The first iteration is when the loop buffer records the instructions that the body of the loop contains. Once, the loop is recorded in the loop buffer, for the rest of the loop iterations, the instructions are fetched from the loop buffer instead of the program memory. In the last iteration, the state-machine detects that the register *LC* is "1" and sets the connections inside of the Instruction Memory Organization such that subsequent instructions are fetched only from the program memory. During the execution of non-loop parts of the code, instructions are fetched directly from the program memory. Figure 3.4 depicts the Instruction Memory Organization with the inclusion of the loop buffer concept.

Because the majority of the signal and image processing applications are dominated by instructions of small loops with 32 or fewer instructions, a loop buffer size of 32 instruction words is utilized. Our implementation of the loop buffer is a flip-flop array of 32 instruction words of 16 bits. The choice of a flip-flop implementation is due to the energy reduction of using flip-flops instead of SRAM for small memory sizes [22].

The state-machine is the element that controls the connections that are inside of the

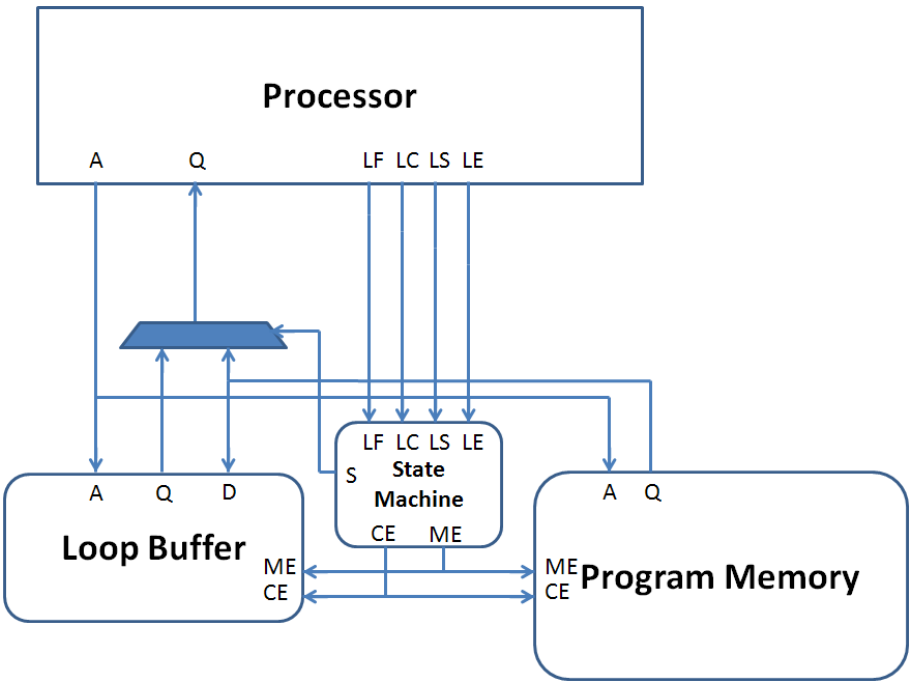


Figure 3.4: Instruction Memory Organization interface.

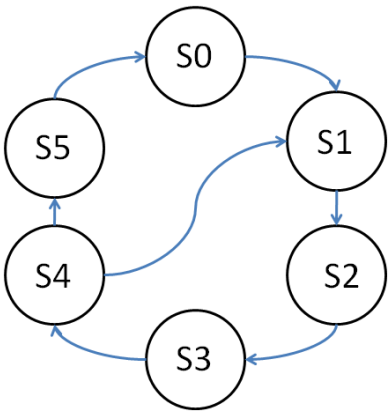


Figure 3.5: State-machine.

Instruction Memory Organization. It has 6 states in order to control loop buffer behavior:

s0 Initial state.

s1 Transition state between *s0* and *s2*.

s2 State where the loop buffer is recording the instructions that the program memory supplies to the processor.

s3 Transition state between *s2* and *s4*.

s4 State where only the loop buffer is the component that supplies the instructions to the processor.

s5 Transition state between the state *s4* and the initial state *s0*.

Figure 3.5 shows the state-machine diagram. The transitions states (*s1*, *s3*, *s5*) are necessary in order to give the control of the instruction supply from the program memory to the loop buffer and vice-versa. The transition between *s4* and *s1* is necessary because the body size of a loop can change in real-time (i.e. in a loop body we have if-statements or function calls). In order to check in real-time that the loop body size does not change, a tag of 1-bit is added to each address space. Hence, a variable composed with these tags is used by the state-machine. When a tag is "1", the relative address space is already written, otherwise, the tag is "0". This bit is always checked in state *s4*. The state-machine is completely implemented in VHDL.

Along this Section, the operation of the loop buffer concept was explained in detail. Besides, an implementation of the loop buffer concept in a central loop buffer architecture for single processor organization was presented. In order to mimic multiple loop buffer architectures with shared loop-nest organization and distributed loop buffer architectures with incompatible loop-nest organization several synthesis of different loop buffer sizes were performed.

3.3 Simulation Methodology

The first step in this methodology is to map the application to the system architecture. With this step, we set how the application receives the input data and how it generates the output data. The second step is to simulate the mapped application on the processor in order to check the correct functionality of the system. For that purpose, an Instruction-Set Simulator (ISS) from Target Compiler Technologies is used. Once the correct functionality of the application is checked, VHDL files of the processor architecture are automatically generated using the HDL generation tool from Target Compiler Technologies. Because of the HDL generation tool only generates the interfaces of the memories in the design, the Data Memory Hierarchy and the Instruction Memory Organization had to be added in order to build the whole system.

When every component of the system architecture has been built in RTL level, the design is then synthesized using a 90 nm Low Power TSMC library. In this design, a frequency of 100 MHz is fixed and clock gating is used whenever possible. After the synthesis, place and route is performed using Encounter (Cadence tool [25]). After place and route, it is necessary to generate a VCD (Value Change Dump) file for the time interval of the netlist simulation. These files contain the information of the activity of every net and every component of the whole system. As a final step, the average power consumption information is extracted with Primetime (Synopsis tool [26]). For both applications, the time interval given to create the VCD file corresponds to the execution time to process an input data frame.

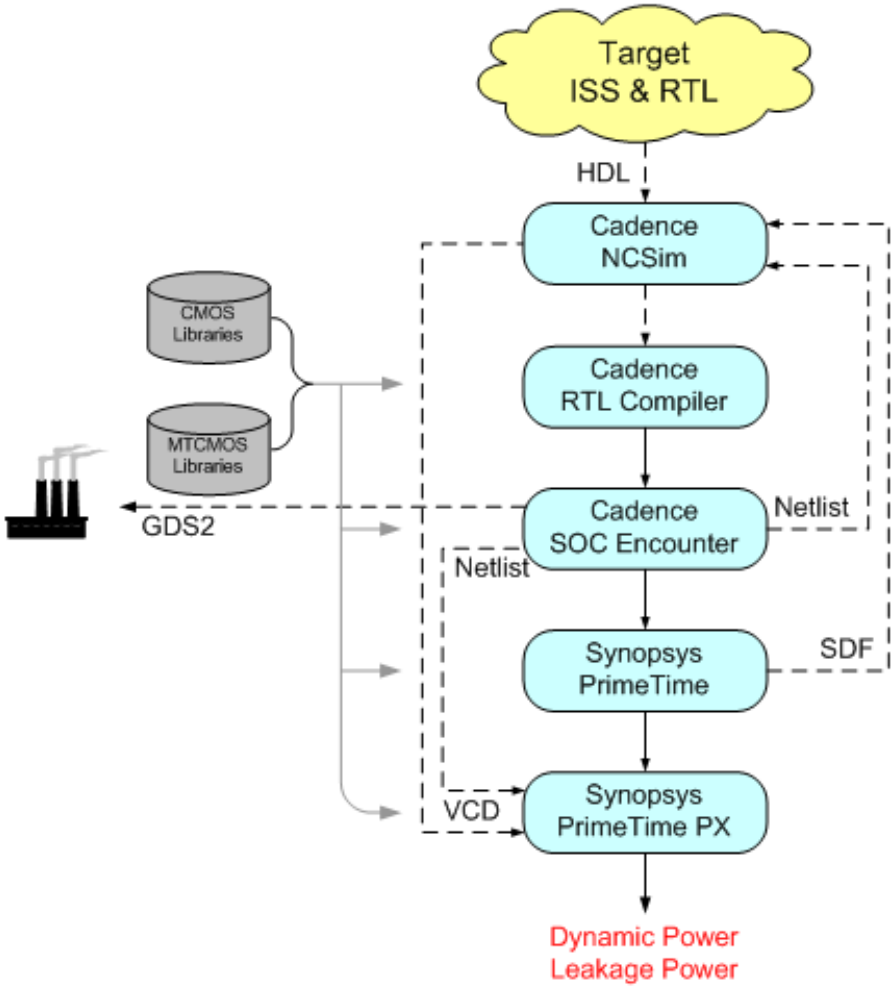


Figure 3.6: Simulation Methodology.

3.4 Design Space Exploration

In order to perform an energy design space exploration of the loop buffer concept based on the classification presented in Chapter 2, synthetic benchmarks are developed to show the energy trends of these architectures.

3.4.1 Synthetic Benchmarks

Synthetic benchmarks are helpful in order to perform an energy design space exploration of the loop buffer concept. The synthetic benchmarks presented in this work mimic loops, found in real embedded system applications, of different size and number of iterations. In order to perform the energy design space exploration, different architectural models have to be built based on the architecture described in Section 3.2.

The parameters to construct loops for the synthetic benchmark are base on Reference [23], and include the loop body size and the number of iterations of the loop. Loop body size ranges from 1 to 32 instruction words. On Reference [23], a study on the loop behavior of embedded applications demonstrates that 77% of the execution time of an application is spent in loops with 32 instructions or less. Moreover, based on the same Reference, a limit of 32000 iterations is selected. Indeed in Reference [23], it is demonstrated that 84% of the execution time is spent in loops with 32000 iterations or less. All the loops that form the synthetic benchmarks share these parameters and their limits.

In order to have precision in the body sizes of the loops, the synthetic benchmarks are implemented in assembly. The assembly instruction words as well as the operands that are present in each loop are randomized. This is different from reality where some correlation is present in these instruction bits, but for the purpose of our loop buffer experiment, these correlations are not that relevant, so they can be ignored here.

For central loop buffer architecture for single processor organization, a synthetic benchmark with sequential loops is developed. The loops that form this synthetic benchmark have a variation in loop body size from 1 to 32 instruction words, and a variation in loop iterations from 1 to 32000 iterations. The central loop buffer size is fixed to 32 instructions words.

In order to mimic the behavior of a multiple loop buffers architecture with shared loop-nest organization, an architectural model based on Reference [3] is used. In this model, the loop buffer concept is distributed for each functional unit of the architecture with a single loop controller. In order to mimic the behavior of loops in such types of architectures (still using the centralized loop buffer presented in Section 3.2), the loop buffer size is also changed (besides changing the loop body size and number of iterations). An architecture of 2 loop buffers is assumed in our experimental framework. The benefits in terms of energy come from the possibility of running loops in parallel, where the loop buffer sizes are tuned to the loop body sizes.

For the case of the distributed loop buffer architecture with incompatible loop-nest organization the synthetic benchmark is based on Reference [17]. Also in this model, the loop buffer concept is distributed but now with distributed loop controllers. Like in the previous architectural model, in order to simplify the energy analysis of these architectures, a distributed architecture that has only 2 loop buffers is selected. In this case, the synthetic benchmark has the same characteristics as the synthetic benchmark

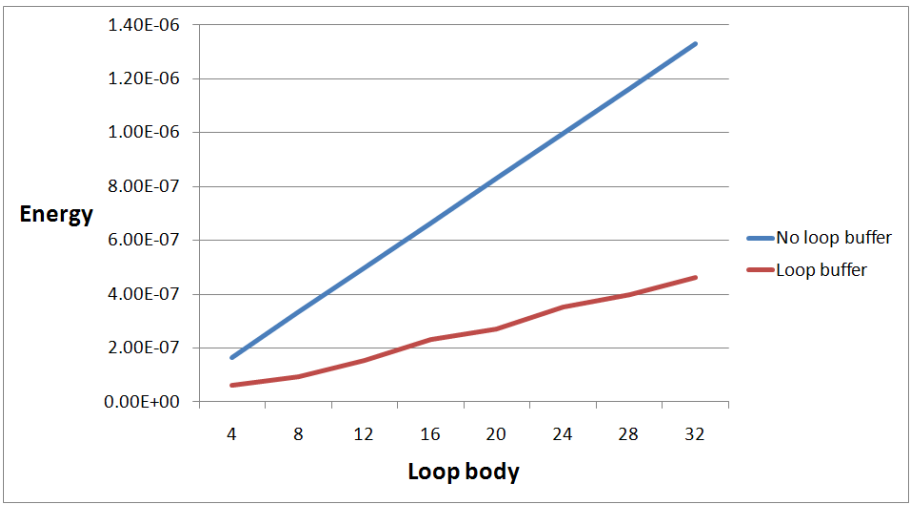


Figure 3.7: Energy variation in Instruction Memory Organization. Loop body size variation based on number of instructions.

related with the multiple loop buffers architecture with shared loop-nest organization, but in this case the instruction level parallelism is improved by the introduction of distributed loop controllers. This enhancement allows to this architecture to run in parallel several loops that have incompatible conditions in their headers.

3.4.2 Energy analysis based on synthetic benchmarks

The energy analysis of the synthetic benchmarks, i.e., the results and estimations, is split in each one of the set of architectural models contained in the classification presented in Chapter 2.

For central loop buffer architecture for single processor organization, the synthetic benchmark with sequential loops is executed on an architecture with a central loop buffer size of 32 instructions words. Figure 3.7 plots the energy consumption as a function of the loop body size for a fixed number of iterations (i.e., 4000). This Figure shows that the energy savings using the loop buffer are directly proportional to the loop body size. This behavior is because the number of accesses redirected to the loop buffer is increased with the loop body size. Figure 3.8 shows that the energy savings are also directly proportional to the number of iterations of the loop. In this Figure, the loop body size is fixed to 4 instruction words. In Figure 3.8, it is also possible to see that the energy savings tend to be larger for larger number of iterations.

Figure 3.9 shows the energy improvements as a two-dimensional function of loop body size and number of iterations. From this last picture, we can conclude that the energy savings that we can achieve by introducing a central loop buffer, is directly related with the loop body size and the number of iterations. The energy saving depict in Figure 3.9 represents $68\% - 74\%$ of reduction of the energy related with the Instruction Memory Organization. This is based on the assumption that all the execution time is spent in loops which is not realistic.

For multiple loop buffer architecture with shared loop-nest organization, the synthetic

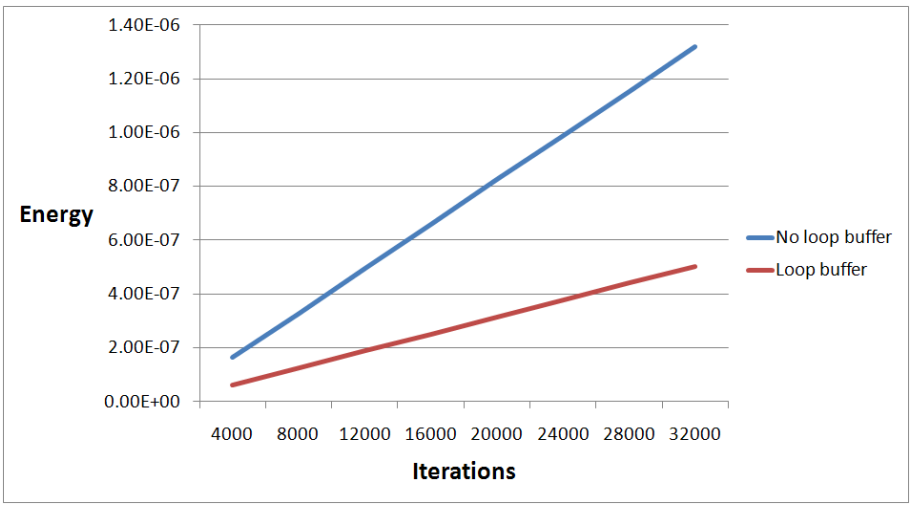


Figure 3.8: Energy variation in Instruction Memory Organization. Number of iterations variation based on number of instructions.

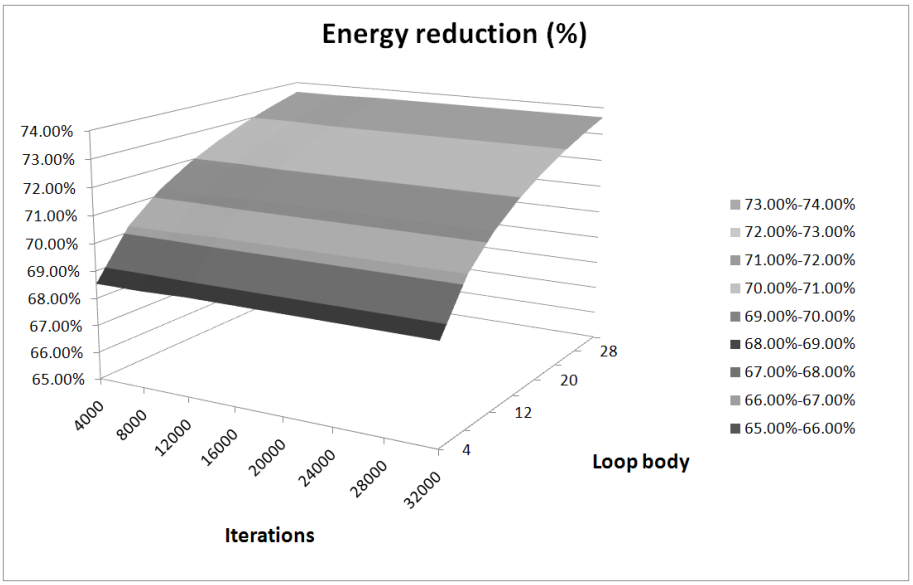


Figure 3.9: Energy improvements in Instruction Memory Organization between system architectures.

benchmark used has loops that can be executed sequentially or in parallel. As multiple loop buffers architectures with shared loop-nest organization are architectural enhancements that increase the performance efficiency of the system by reducing the execution time of an application, our analysis will be focused in the benefits of the loop parallel execution. Table 3.1 presents the power consumptions of several loops, which have different loop body sizes, when they are executed from several loop buffers that differ in loop buffer sizes. A different power consumption for the same loop buffer size configuration is observed depending of the loop body size of the loop that it is running on it. This difference in power consumption is due to the fact that for some configurations the loops are fetched from the program

Table 3.1: Power consumptions [W] with different loop buffer sizes.

Loop body size	Loop buffer size		
	4	16	32
4	$1.02 * 10^{-04}$	$1.72 * 10^{-04}$	$3.73 * 10^{-04}$
16	$1.05 * 10^{-03}$	$1.72 * 10^{-04}$	$3.73 * 10^{-04}$
32	$1.05 * 10^{-03}$	$1.10 * 10^{-03}$	$3.73 * 10^{-04}$

memory instead of the loop buffer (e.g when a loop of 32 instructions words is running on an architecture based on a loop buffer which size is 4 instructions words).

We have to assume that in the body of an application there are 2 loops sequentially coded, where the first loop has a loop body size of 4 instructions words, and the second loop has a loop body size of 32 instruction words. Both loops have 4000 iterations in order to mimic a single loop controller. This application is running on an architecture based on 2 distributed loop buffers with different size (4 and 32 instruction words respectively). From Table 3.1, it is possible to estimate that the energy consumed by this architecture is $E_{MLB} = E_{lb4LB4} + E_{lb32LB32} = (1.02 * 10^{-04} * 4000 * 10^{-8}) + (3.73 * 10^{-04} * 4000 * 10^{-8}) = 1.90 * 10^{-07} J$, where E_{lb4LB4} is the energy that a loop of 4 instruction loop body size consumes in a loop buffer with a 4 instruction words size, and $E_{lb32LB32}$ is the energy that a loop of 32 instruction loop body size consumes in a loop buffer with a 32 instruction words size. If we compare this value with the energy consumed by a central loop buffer architecture which has a total energy consumption $E_{CLB} = E_{lb4LB32} + E_{lb32LB32} = (3.73 * 10^{-04} * 4000 * 10^{-8}) + (3.73 * 10^{-04} * 4000 * 10^{-8}) = 2.98 * 10^{-07} J$. The energy benefit of the multiple loop buffer architecture with shared loop-nest organization in relation with the central loop buffer architecture is $E_{savings} = E_{MLB} - E_{CLB} = 1.08 * 10^{-07} J$, shown a energy reduction of 36.33%.

If our configuration, instead of having different loop buffer sizes, has 2 loop buffers with the same size (e.g., 32 instruction words), based on the previous energy analysis we can conclude that with the introduction of parallelism no energy savings can be achieved, because in this case: $E_{MLB} = E_{lb4LB32} + E_{lb32LB32} = E_{CLB}$. Therefore, the energy savings related with the use of multiple loop buffer architectures with shared loop-nest organization come from the tuning of the loop buffer sizes of the loop buffer architecture based on the loop sizes of the loops that form the application.

For distributed loop buffer architectures with incompatible loop-nest, the synthetic benchmark has the same characteristics as the synthetic benchmark related with multiple loop buffers architecture with shared loop-nest organization, but now the assumption is that in the body of an application there are 3 loops sequentially coded, where the loop body sizes are 4, 16 and 32 instruction words respectively, and its number of iterations are 4000, 4000 and 8000 respectively. This application is running on an architecture based on 2 distributed loop buffers with different size (16 and 32 instruction words respectively). From Table 3.1, it is possible to estimate that the energy consumed by this architecture is $E_{DLB} = E_{lb4LB16I4000} + E_{lb16LB16I4000} + E_{lb32LB32I8000} = (1.72 * 10^{-04} * 4000 * 10^{-8}) + (1.72 * 10^{-04} * 4000 * 10^{-8}) + (3.73 * 10^{-04} * 8000 * 10^{-8}) = 4.38 * 10^{-07} J$, where $E_{lb4LB16I4000}$

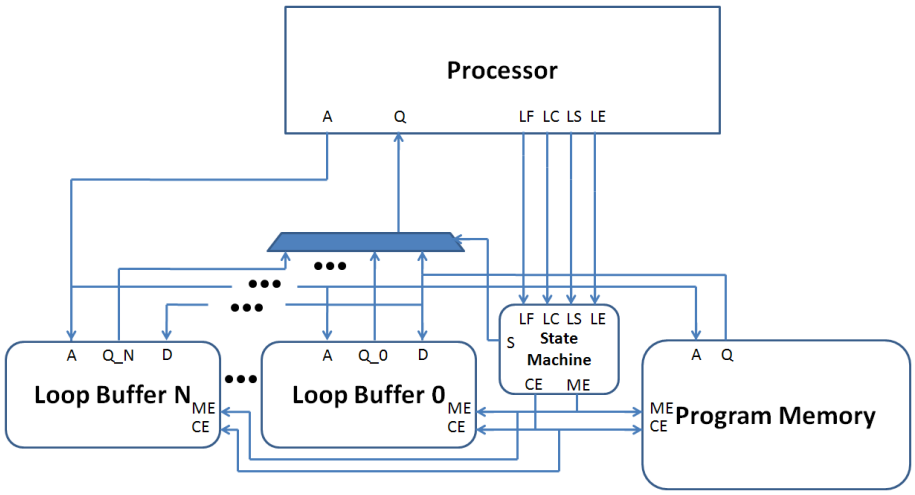


Figure 3.10: Instruction Memory Organization interface for a multiple loop buffer architecture.

is the energy that a loop of 4 instruction loop body size and 4000 iterations consumes in a loop buffer with a 16 instruction words size, $E_{lb16LB16I4000}$ is the energy that a loop of 16 instruction loop body size and 4000 iterations consumes in a loop buffer with a 16 instruction words size, and $E_{lb32LB32I8000}$ is the energy that a loop of 32 instruction loop body size and 8000 iterations consumes in a loop buffer with a 32 instruction words size. Comparing this value with the energy consumed by a central loop buffer architecture, which is $E_{CLB} = E_{lb4LB32I4000} + E_{lb16LB32I4000} + E_{lb32LB32I8000} = (3.73 * 10^{-03} * 4000 * 10^{-8}) + (3.73 * 10^{-03} * 4000 * 10^{-8}) + (3.73 * 10^{-04} * 8000 * 10^{-8}) = 4.48 * 10^{-06} J$, the energy benefit of the distribute loop buffers architecture in relation with the central loop buffer architecture is $E_{savings} = E_{DLB} - E_{CLB} = 4.04 * 10^{-06} J$, shown a energy reduction of 90.2%.

Due to that fact, distributed loop buffer architectures with incompatible loop-nest organization can handle compatible and incompatible loops, an extra performance benefit is achieved if this architectures are compared with multiple loop buffer architecture with shared loop-nest organization. From the energy consumption point of view, our energy analysis shows that the use of distributed loop buffer architectures with incompatible loop-nest instead of multiple loop buffer architecture with shared loop-nest do not provide energy savings, because $E_{DLB} = E_{lb4LB16I4000} + E_{lb16LB16I4000} + E_{lb32LB32I8000} = E_{MLB}$. Therefore, only performance improvements can be achieved using distributed loop buffer architectures with incompatible loop-nest instead of multiple loop buffer architecture with shared loop-nest.

3.4.3 Conclusions

Based on a novel classification of architectures and architectural enhancements based on the use of loop buffer concept, a design space exploration of the loop buffer concept from energy consumption point of view was performed. This design space exploration was focused on different architecture variants based on the loop buffer concept, and their energy impact on different application scenarios.

Gate-level simulations demonstrate that the energy savings that can be achieved introducing the loop buffer in a system, are directly related with the loop body size as well as the number of iterations of the loops. An energy reduction of $68\% - 74\%$ of the total energy budget of the system can be achieved based on these loop characteristics. Besides, the energy savings related with the use of multiple loop buffer architectures with shared loop-nest organization are due to the tuning of the loop buffer sizes of the loop buffer architecture based on the loop sizes of the loops that form the application. The comparison between distributed loop buffer architectures with incompatible loop-nest and multiple loop buffer architectures with shared loop-nest shows that only performance improvements can be achieved using distributed loop buffer architectures with incompatible loop-nest instead of multiple loop buffer architectures with shared loop-nest.

3.5 Case Studies

Based on the loop profiling presented in Table 3.2 the different configurations presented in Table 3.3 are selected by simply taking the maximum loop body size and chop it by the granularity of the smaller loop body size. In every Table presented in this paper the names *Initial*, *SCLB* and *BCLB* represent the Initial system architecture, the system architecture using a Single Central Loop Buffer organization, and the system architecture using a Banked Central Loop Buffer organization respectively.

3.5.1 Case study 1

Heart Beat Detection algorithm

The biomedical application, used as benchmark in our experimental framework, is a HBD (Heart Beat Detection) algorithm based on a previous algorithm developed by Romero et al. [18]. This algorithm uses the CWT (Continuous Wavelet Transform) [28] for automatic heart beat detection. According to Reference [18], the QRS complex is the part of the ECG signal that represents the greatest deflection from the baseline of the signal. Within the QRS complex, the R-wave ideally represents the positive peak. Therefore, many QRS detection algorithms, like the one described in this Subsection, try to detect the R-peak within the QRS complex. However, the technique used in this algorithm could also be used to detect P and T waves within the ECG signal. Figure 3.11 shows the P, Q, R, S and T waves on an ECG signal.

The version of the algorithm used is an optimized C-language version for embedded systems. This algorithm does not require pre-filtering, and it is robust against interfering signals under ambulatory monitoring conditions. The algorithm processes an input data frame of 3 seconds, that includes 2 overlaps of 0.5 seconds each with consecutive frames in order to not loose data between frames. The flowchart of the algorithm is shown in Figure 3.12, and it is described in [18] as follows:

1. The ECG signal is analyzed within a window of 3 seconds. First, the CWT is calculated over this interval. Right after the computation of the CWT, a mask is applied to remove edge components of the result, which is set to be four times the scale of the wavelet.

Table 3.2: Total percentage of execution time of the different sets of loops contained in the applications.

	Loop body size (words) $64-16$	Loop body size (words) $16-8$	Loop body size (words) ≤ 8
Heart Beat Detection Algorithm General-purpose processor	0.62%	6.73%	92.65%
Heart Beat Detection Algorithm Optimized processor	86.74%	11.14%	2.12%
	Loop body size (words) $32-16$	Loop body size (words) $16-8$	Loop body size (words) ≤ 8
Advanced Encryption Standard Algorithm General-purpose processor	2.09%	26.27%	46.49%
Advanced Encryption Standard Algorithm Optimized processor	93.35%	1.02%	5.63%

Table 3.3: Configurations of the experimental framework.

	Initial	SCLB	BCLB
Heart Beat Detection Algorithm General-purpose processor	No loop buffer architecture	8 words	8 banks of 8 words
Heart Beat Detection Algorithm Optimized processor	No loop buffer architecture	64 words	8 banks of 8 words
Advanced Encryption Standard Algorithm General-purpose processor	No loop buffer architecture	8 words	4 banks of 8 words
Advanced Encryption Standard Algorithm Optimized processor	No loop buffer architecture	32 words	4 banks of 8 words

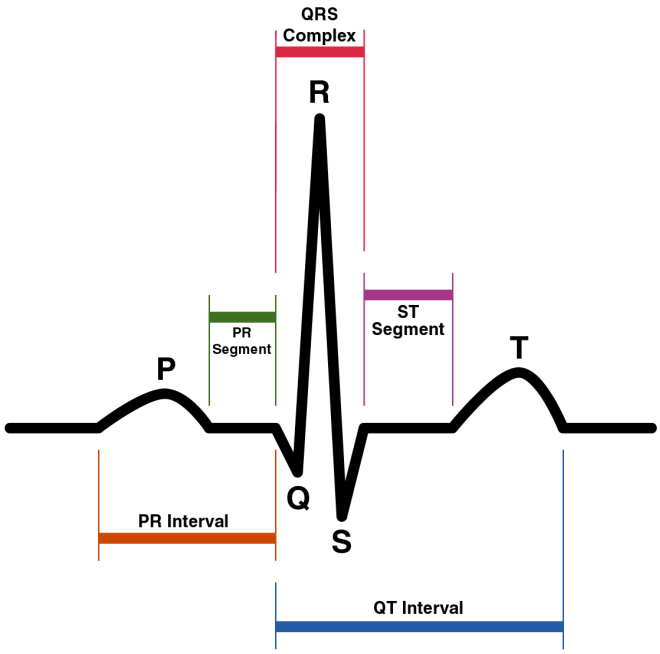


Figure 3.11: P, Q, R, S and T waves on an ECG signal.

2. In the second step, the modulus maxima of the CWT is extracted. The square of the modulus is taken in order to emphasize the differences between coefficients. In addition, the maxima lines below a chosen threshold are ignored.

3. The results from the previous step are taken to be possible R wave peaks. In order to separate the different peaks from each other, all modulus maxima points within intervals of 0.25 seconds are analyzed in turn as search intervals. In every search interval, the point with the maximum coefficient value is selected as R wave peak. The assumption in this step is that all the coefficients within each search interval of 0.25 seconds are due to the same QRS complex. Therefore, only the maximum point is chosen.

4. Finally, the algorithm finds the exact location of the peak in time-domain, based on the results from the previous step. This is done by calculating the mean value of 0.10 seconds before and after the point detected in the wavelet domain. Then the point within the same interval of 0.2 seconds is located, where the signal is furthest from the mean. This step is necessary, because the modulus maxima line does not necessarily point to the exact location of the R wave peak in time domain.

The input data of this algorithm is a signal which contains an electrocardiogram (ECG) from MIT/BIH database [8]. The output data of the algorithm is the time position of the heart beats included in the input data frame. The testing of this optimized algorithm resulted in a sensitivity of 99.68% and a positive predictivity of 99.75% on the MIT/BIH database.

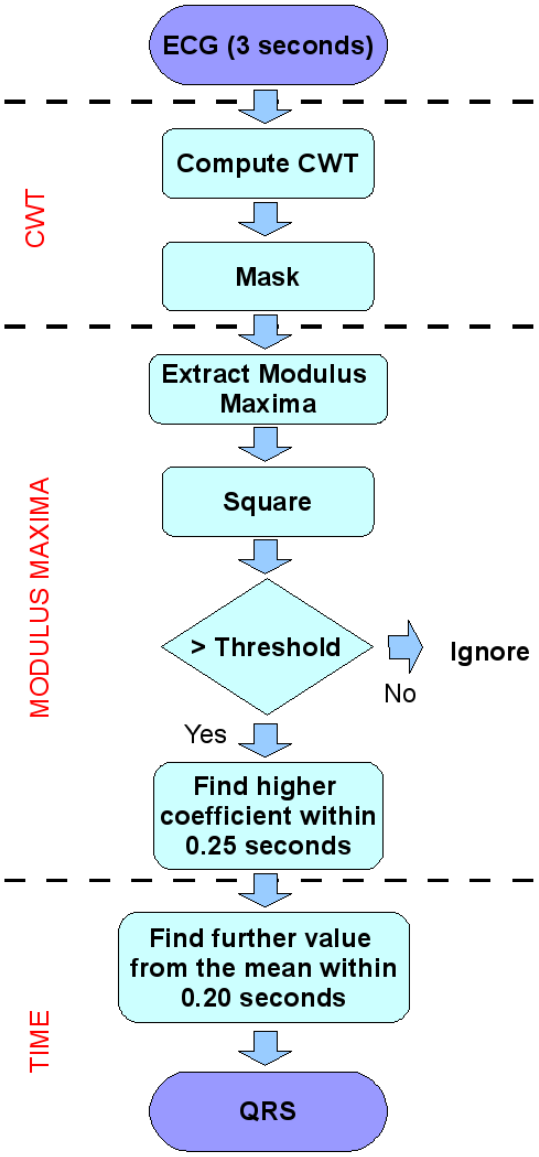


Figure 3.12: Diagram of the Heart Beat Detection algorithm.

Optimized processor for Heart Beat Detection algorithm

The optimized processor for Heart Beat Detection algorithm, which was implemented by Yahya H. Yassin [27], is based in the processor architecture presented in section 3.1. This Subsection presents the modifications and optimizations that were performed in order to built an ASIP processor optimized for this algorithm presented in Subsection 3.5.1.

To design an ASIP processor optimized for a specific application, a deep analysis has to be performed. From this analysis, a critical loop was detected in this specific application. Figure 3.13 shows the critical loop. As it can be seen, this critical loop is contained in the convolution step within the Continuous Wavelet Transform. A shift operation performed in this loop forces some variables to be defined as *long*. If the processor used for this application is the processor presented in Subsection 3.1, additional instructions have to be executed in order to handle variables of 32 bits, hence making longer the execution time of the application. Using a processor with a 32-bit data-path, it is possible to decrease this execution time. There is always a trade-off between complexity in the processor and its energy consumption. However, for this specific scenario, it is a benefit to have a processor of 32-bit data-path. Results presented in Tables 3.4 and 3.5 demonstrate that the benefit from the reduction in the execution time compensates the penalty due to the complexity of the use of a 32-bit data-path processor instead of a 16-bit data-path processor. Due to the decision to have a 32-bit data-path processor, an extension of the addressing mode and the word data type of the processor were required (from 16-bit addressing to 32-bit addressing). Besides, these changes require a change in all the instructions that are related to immediate values.

After this modification, several optimizations related with the main critical loop of the application were applied. In each one of the following paragraphs, these optimizations are described.

By analyzing the assembly code presented in Figure 3.13, it is possible to see that in the main critical loop, a signed multiplication is performed, and after its execution, its result is accumulated in a temporally variable. The highlighted part of Figure 3.13 shows all instructions involved to execute the corresponding C code for the critical loop. The execution of these 6 assembly instructions is 72% of the execution time of the application according to the profiling information. Hence, reducing these set of instructions results in a shorter execution time for the application. Therefore, the MUL unit is modified to multiply two signed integers, and accumulate without shifting the result of the multiplication. This is done in the primitives definition and generation language of the processor. This optimization saves energy in two ways: reducing the complexity of the MUL unit and the execution time of the application.

The load operations related to the custom MUL operation are combined in a customized instruction to be executed in parallel. However, in the general-purpose processor it is only possible to load and store data from the same memory once per pipeline stage. To solve this bottleneck, the main data memory is split in two identical data memories, DM (Data Memory) and CM (Constant Memory). The assignment of variables to a specific memory is done by specifying this in the C code directly. It is possible to perform the load instructions in parallel by assigning the variables correctly. In order to access two memories in parallel, another address generator (ag2) is created such that the load and store operations from the DM and CM memories could be performed within the same pipeline stage. This is

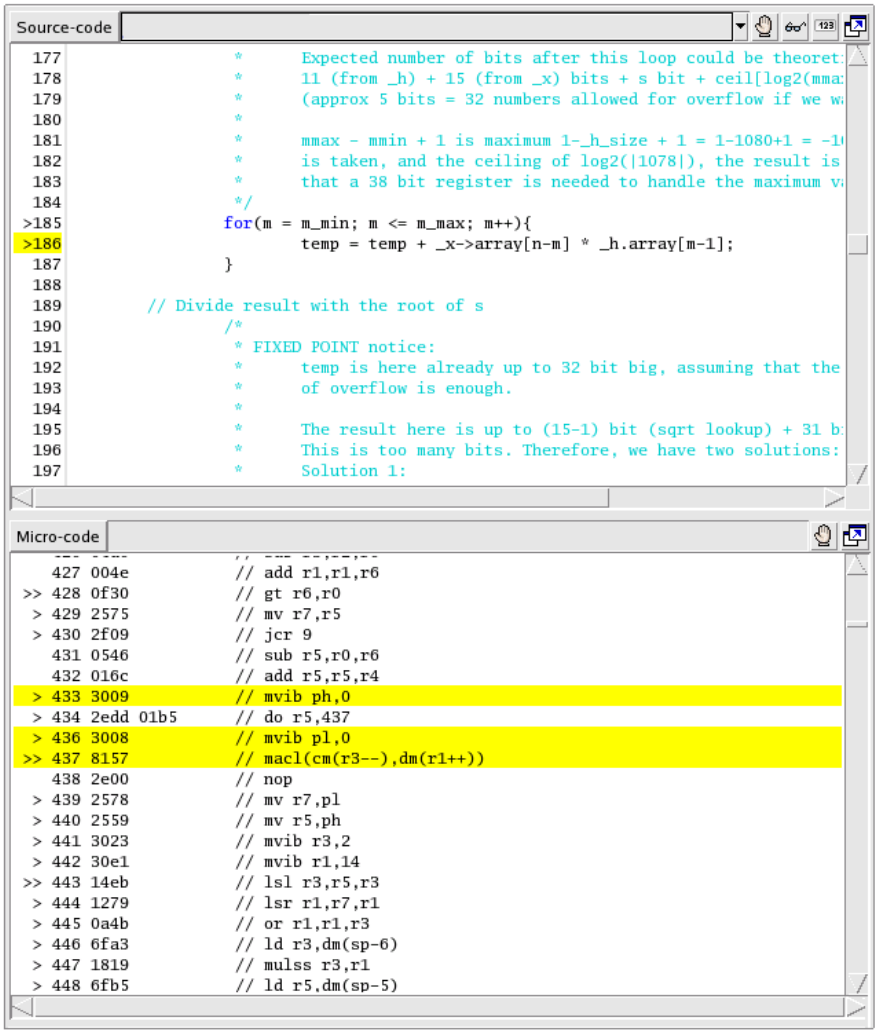


Figure 3.13: Critical loop in the Heart Beat Detection algorithm.

done through separate ports of the register file as shown in Figure 3.14.

As the input registers of the MUL unit can be loaded directly, a new modification can be performed. However, doing this requires another pipeline stage. The parallel load and MUL instructions are then combined by first adding another pipeline stage, and creating a custom instruction that integrates both MUL and parallel load instructions. The MUL instruction is executed in the second pipeline stage, while the parallel load instruction executes in the first pipeline stage. Instead of assigning the two multiplicands to the multiplier from registers, additional pipes were created. In this case, a pipe is used to delay a signal one pipeline stage. After this last modification, the MUL operation included in the main critical loop of the application is performed using only one assembly instruction.

In a similar way as the MUL operation, another critical loop is optimized by combining a load, select and equal instructions to be executed in parallel. This instruction was created adding the functionality from the equal and select instruction, and combining them together with a normal load operation. As it can be seen in Figure 3.14, there are

two ALUs working in parallel, ALU and ALU 2. The second ALU is created for the select operation.

All optimizations and modifications presented in this Subsection result in a new processor architecture, as it is shown in Figure 3.14. Basically, an address generator and a second ALU are added, in addition to some pipes and some ports.

In order to handle ECG signals sampled at 1 KHz, the data memory required by this processor architecture is a DM memory with a capacity of 8k words/32 bits, and a CM memory with a capacity of 8k words/32 bits. On the other hand, the program memory required is a memory with a capacity of 1k words/20 bits. The PC (Program Counter unit) is modified to handle instruction words that use 32-bit immediate values.

Heart Beat Detection Algorithm Energy Analysis

In Subsection 3.5.1 the HBD algorithm was described. Here, an energy analysis is performed by comparison between the general-purpose processor (see Section 3.1) and the processor optimized for this algorithm (see Subsection 3.5.1). In each of these scenarios, different configurations of the loop buffer concept are tested.

A 100MHz system frequency is fixed to meet time requirements. At this system frequency, this application running on the general-purpose processor spends 462 cycles in order to process an input sample contained in the data frame. However, if this algorithm is running on the processor optimized for this algorithm, the number of cycles in order to process an input sample contained in the data frame is 11 cycles.

Profiling information of the original and optimized applications are depicted in Figures 3.16 and 3.15. These Figures present the number of cycles that this application spent per program counter (PC). Given that the PC is directly related with the program address space, there are regions that are more frequently accessed than others implying the existence of loops.

As we can see, the HBD algorithm is a perfect candidate to perform the energy evaluation, because in it, the execution time of loops, which have less than 32 instructions as body loop, represents approximately 78.62% of the total execution time in the case of the general-purpose processor, and 86.74% in the optimized processor.

The power breakdowns running the HBD algorithm on the general-purpose processor and the optimized processor are shown in Figures 3.17 and 3.18 respectively. In these Figures, it is possible to see how the power distribution changes from a design based on a general-purpose processor to an ASIP design.

In this specific application, with the characteristics presented in previous paragraphs, Table 3.4 and Table 3.5 summarize the power differences between the execution of the same algorithm on the different configurations. These Tables show the dynamic power and the leakage power for all the configurations. The decrease of dynamic power in the system with central loop buffer architecture in relation with the baseline architecture can be explained, because the majority of instructions are fetched from a small loop buffer, reducing the power consumed by the Instruction Memory Organization.

However, having banked central loop buffers brings less power benefit in this application than single central loop buffers due to the power overhead of having complex control

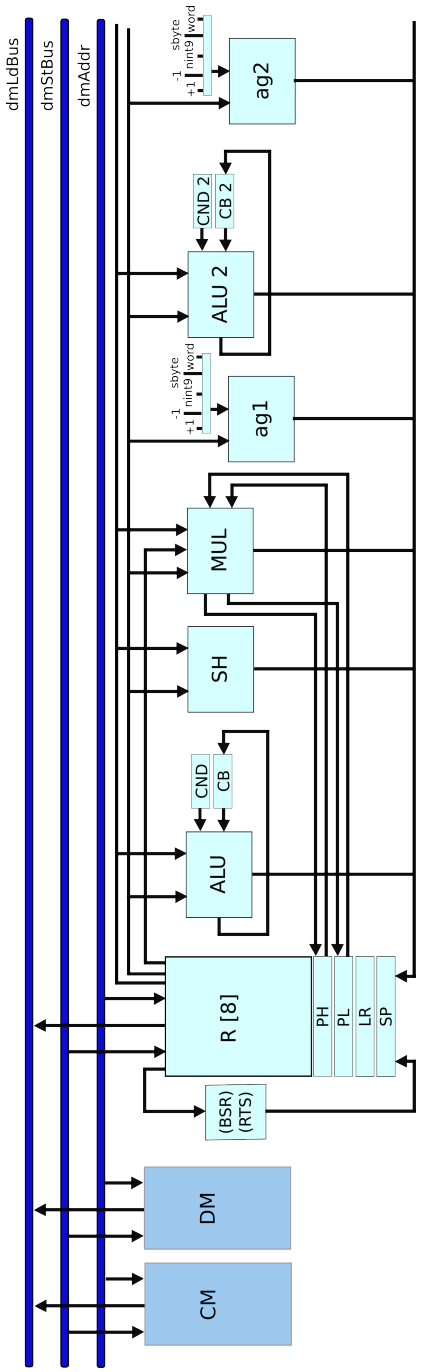


Figure 3.14: Data-path of the optimized processor for the Heart Beat Detection algorithm.

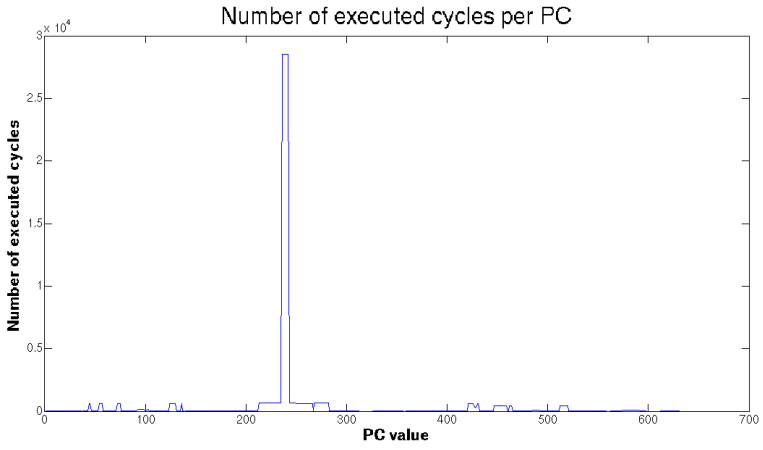


Figure 3.15: Number of cycles per program counter (PC) in the general-purpose system. HBD algorithm.

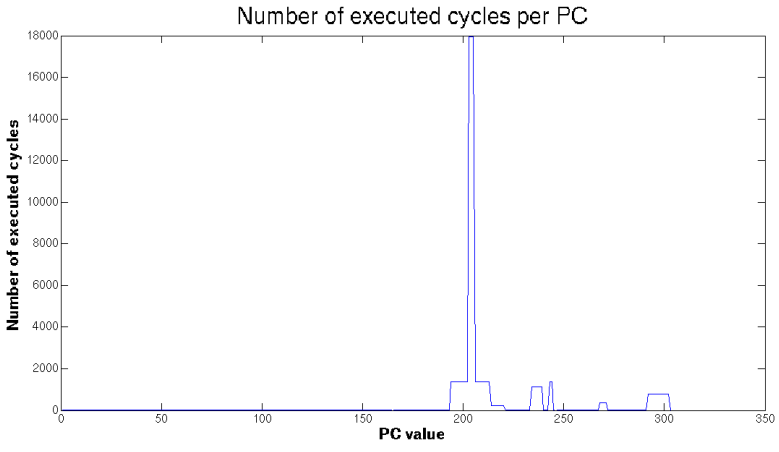


Figure 3.16: Number of cycles per program counter (PC) in the optimized system. HBD algorithm.

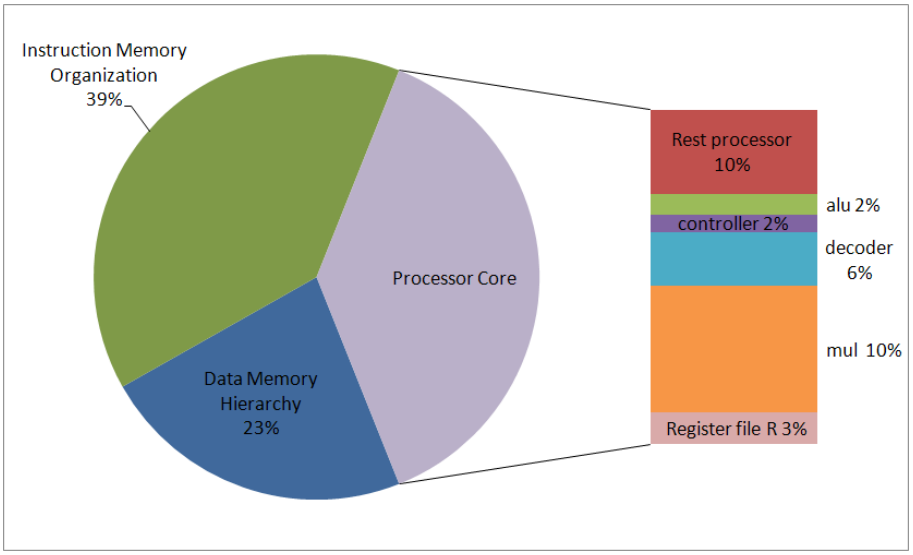


Figure 3.17: Power breakdown for the general-purpose processor running the Heart Beat Detection algorithm.

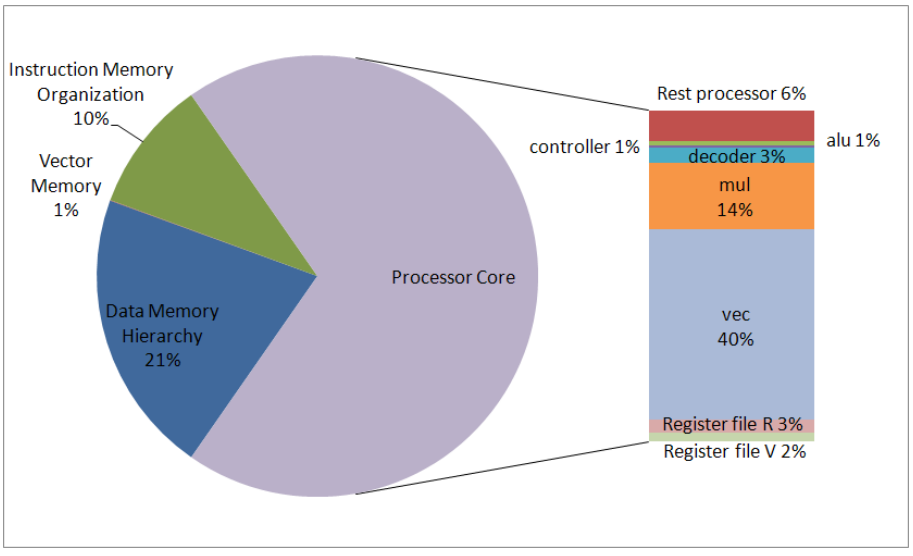


Figure 3.18: Power breakdown for the optimized processor running the Heart Beat Detection algorithm.

Table 3.4: Power consumption [W] of the Instruction Memory Organizations used by the HBD algorithm with the general-purpose processor.

	Dynamic Power [W]	Leakage Power [W]	Total Power [W]
Initial	$4.44 * 10^{-6}$	$0.91 * 10^{-9}$	$4.44 * 10^{-6}$
SCLB	$1.74 * 10^{-6}$	$1.14 * 10^{-9}$	$1.73 * 10^{-6}$
BCLB	$1.97 * 10^{-6}$	$1.47 * 10^{-9}$	$1.97 * 10^{-6}$

Table 3.5: Power consumption [W] of the Instruction Memory Organizations used by the HBD algorithm with the optimized processor.

	Dynamic Power [W]	Leakage Power [W]	Total Power [W]
Initial	$3.57 * 10^{-7}$	$8.46 * 10^{-11}$	$3.57 * 10^{-7}$
SCLB	$1.40 * 10^{-7}$	$1.77 * 10^{-10}$	$1.40 * 10^{-7}$
BCLB	$1.64 * 10^{-7}$	$3.83 * 10^{-10}$	$1.65 * 10^{-7}$

logic, and the lack of different loop sizes which could use the memory banks. On the other hand, the leakage power is increased from the baseline architecture to the system architectures with banked central loop buffers due to the increase in the complexity of the logic. Finally, we can see how considering the total power consumption, the system architecture with a single central loop buffer is the best option design, because it reduces 61% the total power consumption both for the case of the general-purpose processor, and the optimized processor. With this result we can conclude that, to use a single central loop buffer configuration tuned to the loop body size of the loops, that are most used in the application, brings more benefit than a banked central loop buffer configuration. The reason is that the design, which is adapted dynamically to every loop contained in the application, is more complex and the increase in complexity jeopardizes the decrease in power consumption.

3.5.2 Case study 2

Advanced Encryption Standard algorithm

The cryptographic application, used as first case study, is the security mode of operation AES-CCM-32. This mode of operation is based on the AES (Advanced Encryption Standard) algorithm [16] and provides confidentiality, data integrity, data authentication, and replay protection. The message authentication code is 32 bits wide.

AES is a symmetric-key encryption standard in which both the sender and the receiver use a single key for encryption and decryption. The data block length used by this algorithm is fixed to 128 bits, while the length of the cipher key can be 128, 192 or 256 bits, corresponding with the three block ciphers that this standard comprises. The AES algorithm is an iterative algorithm in which the iterations are called rounds, and the total number of rounds can be 10, 12, or 14, depending whether the key length is 128, 192, or 256 bits, respectively. The 128-bit data block is divided into 16 bytes. These bytes are mapped to a 4 x 4 array called the State, and all the internal operations of the AES algorithm are performed on the State. Each byte in the State is denoted by $S_{i,j}$, where $0 < i$ and $j < 5$, and it is considered as an element of Galois Fields, $GF(2^8)$. The irreducible polynomial used in the AES algorithm to construct $GF(2^8)$ field is $p(x) = x^8 + x^4 + x^3 + x + 1$. Figure 3.19 presents the AES encryption process. In the encryption of the AES algorithm, each round except for the final round, consists of four transformations: the SubBytes(), the ShiftRows(), the MixColumns(), and the AddRoundKey(), while the final round does not have the MixColumns() transformation. The decryption is obviously the inverse process of the encryption.

On the other hand, the CCM (CTR-CBC-MAC), which is presented in the NIST Special Publication 800-38C [6], encrypts and authenticates the message, authenticates a set of data which are called associated data (e.g., header, etc.) and assigns a nonce (Number used ONCE) to the payload and the associated data. A nonce is a random or pseudo-random number issued to ensure that old communications cannot be reused in replay attacks. Both communication participants know the value of nonces and update them increasing the number by one after the usage. Depending on the size of the message authentication code it produces (4, 8 or 16 bytes), we have three different variations of AES-CCM: AES-CCM-32, AES-CCM-64, AES-CCM-128.

Due to ultra low power requirements, the proposed algorithm supports only 128-bit key. In addition, only the encryption core of AES algorithm is supported. However, with a very small change in the design, both encryption and decryption can be supported. The input data of this algorithm is a data packet which payload is fixed to 1460 bytes. The output data of the algorithm is a data packet which has the same size than the input data, and where the data included is encrypted.

Optimized processor for Advanced Encryption Standard algorithm

The optimized processor for Advanced Encryption Standard algorithm, which was implemented by Ioanna Tsekoura in [21], is also based in the processor architecture presented in Section 3.2. This subsection presents the modifications and optimizations that were performed in order to built an ASIP processor optimized for the Advanced

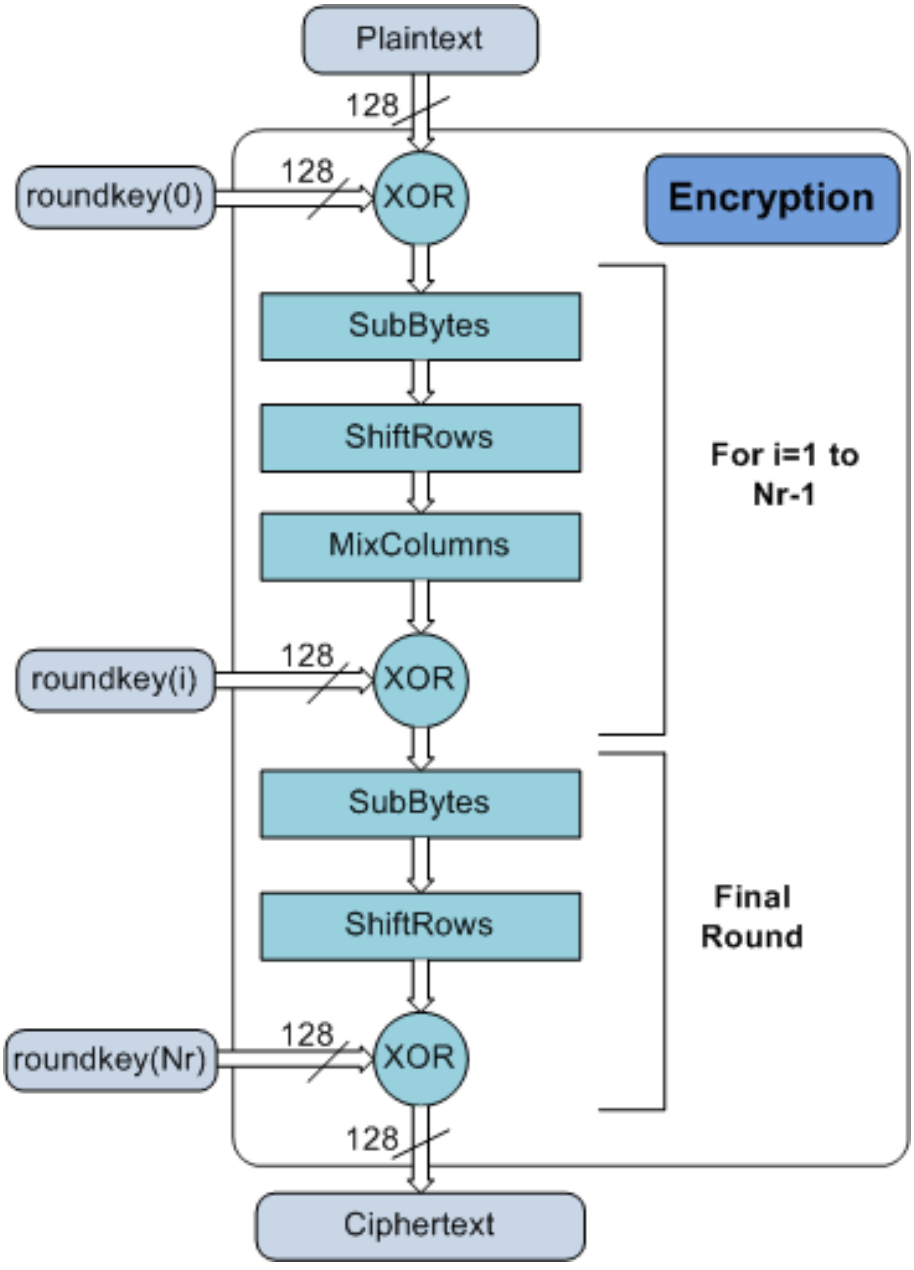


Figure 3.19: Flowchart of the AES algorithm. Encryption process.

Encryption Standard algorithm presented in Subsection 3.5.2.

Analyzing the application, the critical functions were identified and optimized in order to have better performance in terms of clock cycles and memory accesses. Custom techniques were applied to implement more efficient code. The following source code transformations were applied:

- Function Combination. During a function call, the present state (returning address, registers, etc.) has to be saved in the data memory and then retrieved at the end of its execution. Therefore, combining multiple functions in one, it is possible to decrease the memory accesses. The combined functions are not data dependent, therefore we have the possibility to merge them.
- Loop Unrolling. Instead of using a loop construct and an iterator index to perform the same operations at different sets of data, the context of the loop can be repeated multiple times. Although this technique leads to an increase in the code size, it usually also improves the performance, since it eliminates the calculation of the array index based on the loop counter. The loop unrolling is efficiently applied at the 9 rounds of AES, leading to a reduction in clock cycles.

Also, mapping optimization techniques were applied:

- Use of look-up tables. Two of the AES functions are based on Galois Fields, and can be implemented with two approaches: with mathematics and with look-up tables. The first approach is computationally demanding, which means that it needs many execution cycles and accesses in memory, while the second can be demanding in memory area, but it is much faster and does less memory accesses compared to the first one. As this design was focused on power and energy, the look-up table approach was selected.
- Elimination of divisions and multiplications. It is always preferable to substitute computationally demanding operations such as multiplications and divisions by lower overhead instructions. In our applications, all the divisions and multiplications have divisors or multipliers, which are powers of 2. Therefore we replace these operations with right or left shift operations accordingly. Specifically, a division or multiplication with 2n, is equivalent to a right or left shift by n bits accordingly.
- Instruction set extensions. This technique, which is the main advantage of ASIP processors, extends the instruction set of a processor with customized operations for our application. The result is the improvement of the performance of the application mapped on this processor.

In the design of this optimized processor, the structure of the general-purpose architecture is kept intact (16-bit data-path, data memory, register file and ALU), and an extra data-path of 128 bits is added. Figure 3.20 shows both data-paths in the 128-bit ASIP design. The added data-path is connected with a vector memory (VM), a vector register file (V) and a vector unit (Functional Vector Unit). The vector unit includes the AES accelerating operations, as well as the logic and arithmetic instructions that this application requires.

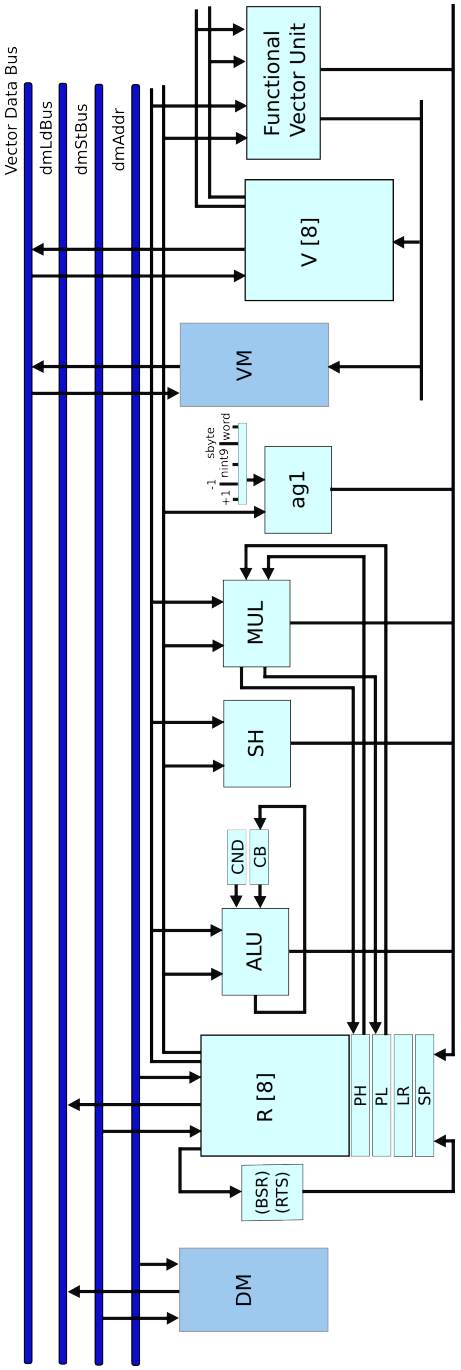


Figure 3.20: Data-path of the optimized processor for the Advanced Encryption Standard algorithm.

One of the advantages of the indicated design approach is the ability to use the larger vector units only when the specific cryptographic domain needs them.

In this processor, the instruction set architecture is extended with one AES accelerating operation: *enc_sched*(). This instruction has 2 inputs, a 128-bit input which can be the State or a Round key, and an integer which indicates the behavior of the *enc_sched*(). The output is 128 bits and contains the State or a Round key, accordingly to the second input.

All optimizations and modifications presented in this Subsection result in a new processor architecture, as shown in Figure 3.20. Basically, an extra data-path of 128 bits is added including a vector memory, a vector register file and a vector unit. In order to handle an input signal of 1460 bytes, the data memory required by this processor architecture is a memory with a capacity of 1k words/16 bits, and a vector memory with a capacity of 64 words/128 bits. On the other hand, the program memory required is a memory with a capacity of 1k words/16 bits. The PC unit in this architecture design is not modified.

Advanced Encryption Standard Algorithm Energy Analysis

In Subsection 3.5.2 the AES algorithm was described. Here, an energy analysis is performed by comparison between the general-purpose processor (see Section 3.1) and the processor optimized for this algorithm (see Subsection 3.5.2). In this analysis, the different configurations of the loop buffer concept are tested too.

In this algorithm, a 100MHz system frequency is also fixed. At this system frequency, this application running on the general-purpose processor spends 484 cycles in order to process an input sample contained in the data frame. However, if this algorithm is running on the processor optimized for this algorithm, the number of cycles in order to process an input sample contained in the data frame is only 3 cycles.

Profiling information of the original and optimized applications are depicted in Figure 3.22 and 3.21 in order to see how the loop distribution changes from one application to the other.

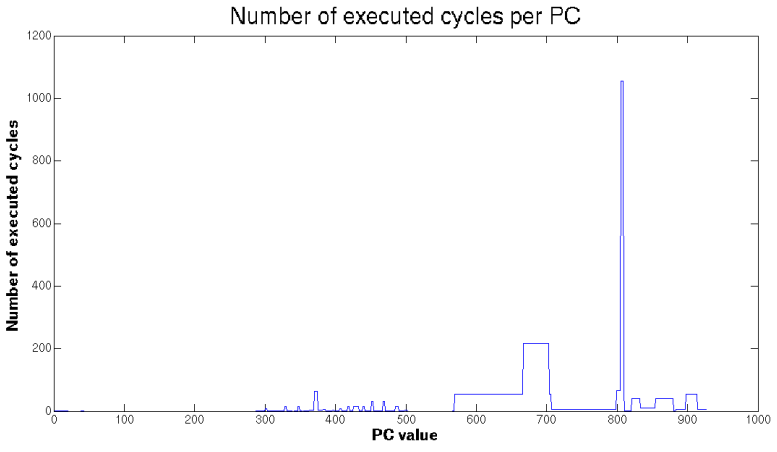


Figure 3.21: Number of cycles per program counter (PC) in the general-purpose system.

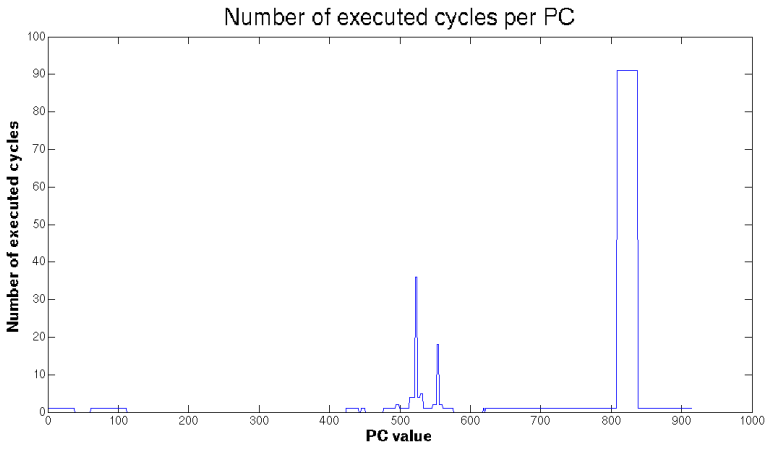


Figure 3.22: Number of cycles per program counter (PC) in the optimized system.

As we can see from the previous Figures, the AES algorithm is a perfect candidate to perform the energy evaluation, because, in it, the execution time of loops, which have less than 32 instructions as body loop represents approximately 14% of the total execution time in the case of the general-purpose processor and 82.23% in the optimized processor.

From Figures 3.23 and 3.24, which present the power breakdown for the experimental framework explained in Section 3.2 running the AES algorithm, it is also possible to see how the power distribution changes from a general-purpose processor to an ASIP processor.

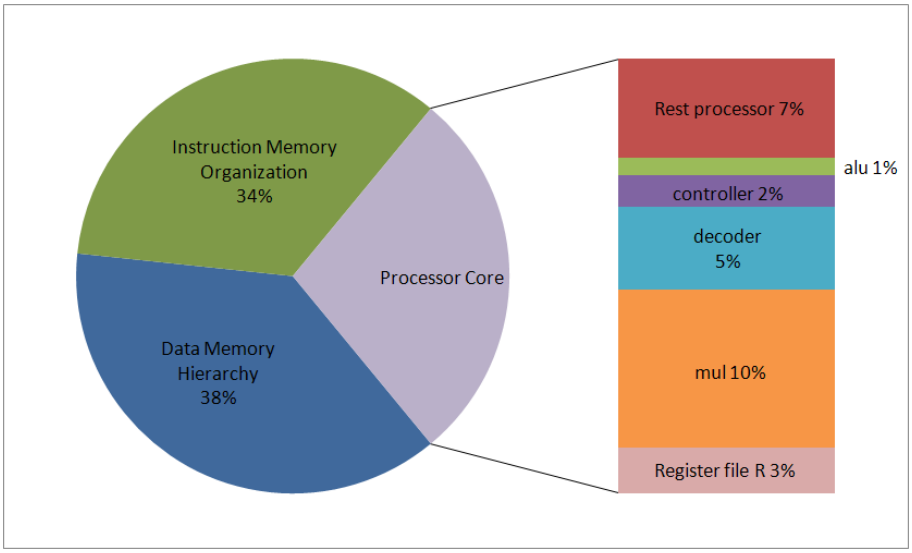


Figure 3.23: Power breakdown for the general-purpose processor running the Advanced Encryption Standard algorithm.

From Table 3.6, we can see how the decrease of dynamic power in the system with single central loop buffer architecture in relation with the baseline architecture can be explained, because some instructions are fetched from a small loop buffer reducing the power consumed by the Instruction Memory Organization. However, we can see that, due to the increased

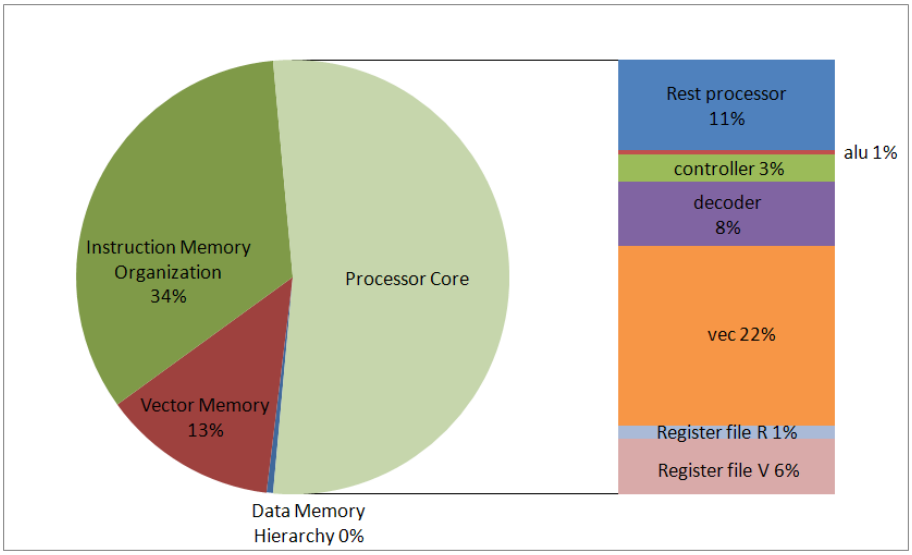


Figure 3.24: Power breakdown for the optimized processor running the Advanced Encryption Standard algorithm.

Table 3.6: Power consumption [W] of the Instruction Memory Organizations used by the AES algorithm with the general-purpose processor.

	Dynamic Power [W]	Leakage Power [W]	Total Power [W]
Initial	$1.81 * 10^{-6}$	$4.32 * 10^{-10}$	$1.82 * 10^{-6}$
SCLB	$1.80 * 10^{-6}$	$5.30 * 10^{-10}$	$1.80 * 10^{-6}$
BCLB	$1.90 * 10^{-6}$	$7.40 * 10^{-10}$	$1.90 * 10^{-6}$

complexity of an Instruction Memory Organization based on banked central loop buffer architectures, the power consumption increases. On the other hand, the leakage power is increased from the baseline architecture to the system architectures with banked central loop buffers, due to the more complex logic. Finally, we can see from the total power consumption of all these configurations that in this scenario the loop buffer concept does not provide significant energy savings. In this system, the total power consumption is only reduced in 0.01% by a central loop buffer configuration.

From Table 3.7 the dynamic power has the same behavior than the system with single central loop buffer architecture. However, in this scenario we can see that the increase in complexity from the an Instruction Memory Organization based on a single central loop buffer architecture to Instruction Memory Organizations based on banked central loop buffer architectures is compensated by the reduction in the total power consumption. This energy savings are due to the tuning of the loop buffer sizes based on the length of the application loops. On the other hand, the leakage power is increased from the baseline architecture to the system architectures with banked central loop buffers, due to

Table 3.7: Power consumption [W] of the Instruction Memory Organizations used by the AES algorithm with the optimized processor.

	Dynamic Power [W]	Leakage Power [W]	Total Power [W]
Initial	$1.20 * 10^{-6}$	$2.11 * 10^{-10}$	$1.20 * 10^{-6}$
SCLB	$8.32 * 10^{-7}$	$4.12 * 10^{-10}$	$8.36 * 10^{-7}$
BCLB	$6.60 * 10^{-7}$	$4.30 * 10^{-10}$	$6.60 * 10^{-7}$

the increase in number of gates. Finally, we can see from the total power consumption of all these configurations, the system architecture with banked central loop buffers is the best option design because it reduces around 45% of the total power consumption in the case of the general-purpose processor. With these results, we can conclude that in this specific scenario, to have a banked central loop buffer configuration, which is design in order to be adapted dynamically to every loop contained in the application, is the best decision design.

After this conclusion, having a look in the profiling information given in Figures 3.15, 3.16, 3.21 and 3.22, we detect why the conclusion from Subsection 3.5.1 differ from conclusion in Subsection 3.5.2. The reason is the distribution of the loops contained in the application and the iterations that each one of them has. Table 3.2 shows how the execution time of loops is more spread for the configuration based on the general-purpose processor than the configuration based on the optimized processor, where 93% of its execution time is in loops of 32-16 words.

Chapter 4

Conclusions

In this manuscript, a novel classification of architectures and architectural enhancements based on the use of loop buffer concept was presented:

- Central loop buffer architecture for single processor organization.
- Multiple loop buffers architecture with shared loop-nest organization.
- Distribute loop buffers with incompatible loop-nest.

Based on this classification, an energy design space exploration of the loop buffer concept was performed. This design space exploration was focused on different architecture variants based on the loop buffer concept, and their energy impact on different application scenarios.

From this energy design space exploration, gate-level simulations have demonstrated that the energy savings that can be achieved introducing the loop buffer in a system, are directly related with the loop body size and the number of iterations of the loops. An energy reduction of $68\% - 74\%$ of the total energy budget of the system can be achieved based on these loop characteristics. Besides, the energy savings related with the use of multiple loop buffer architectures with shared loop-nest organization are due to the tuning of the loop buffer sizes of the loop buffer architecture based on the loop sizes of the loops that form the application. The comparison between distributed loop buffer architectures with incompatible loop-nest and multiple loop buffer architecture with shared loop-nest shows that only performance improvements can be achieved using distributed loop buffer architectures with incompatible loop-nest instead of multiple loop buffer architecture with shared loop-nest.

In order to test these assumptions, as real-life benchmarks, embedded applications mapped on nodes of biomedical wireless sensor networks were used to evaluate the energy reduction obtained using Instruction Memory Organizations based on loop buffer architectures. To evaluate the energy impact, a post-layout simulation was used to have an accurate estimation of parasitics and switching activity. The evaluation was performed using TSMC 90nm Low Power library and commercial memories.

Gate-level simulations have demonstrated that a trade-off exists between the complexity of the loop buffer architecture and the power benefits of utilizing it. This justifies our results, showing that the banked central loop buffer does not always brings benefits. On the other

hand, another conclusion obtained from the results presented in this manuscript is that, if the application has loops with small to medium execution time percentage of the total application execution time, the use of loop buffer architectures in order to bring power benefits to the system should be very carefully evaluated. Using a biomedical application, the energy of the instruction memory organization is reduced by 61% using a single central loop buffer architecture, whereas using a cryptographic application is reduced by 45% using a banked central loop buffer architecture.

Appendix A

Submitted Publications

Energy Efficiency using Loop Buffer based Instruction Memory Organizations

Ready to be submitted to: GLSVLSI 2011

Energy consumption in embedded systems is partially dominated by the consumption of the instruction memory organization. Based on this, any architectural enhancement in this component of the system will produce a significant energy consumption reduction in the total energy budged of the system. Loop buffering is an effective scheme to reduce energy consumption in the instruction memory organization. In this paper, a novel classification of architectures and architectural enhancements based on the use of loop buffer concept is presented. Moreover, based on this classification, an energy design space exploration is performed, focusing on different architecture variants based on the loop buffer concept, as well as on their energy impact on different application scenarios. From gate-level simulations, energy savings of $68\% - 74\%$ of the total energy budget of the system can be achieved for a synthetic benchmark. Based on the energy analysis performed, it is also demostrated that energy savings related with the use of multiple or distributed loop buffer architectures are not related with the instruction level parallelism that they introduces. The energy savings can be achieved adapting the loop buffer sizes of the loop buffer based instruction memory organization to the loop body sizes of the loops that form the application.

Power Impact of Loop Buffer Schemes for Wireless Sensor Nodes

Submitted to: EURASIP Journal "Power-Aware Embedded and Real-Time Systems"

In this paper, the loop buffer concept is applied in real-life embedded application domains widely used in wireless sensor nodes. The loop buffer organizations in which this analysis is focused are the single and the banked central loop buffer architecture. The evaluation

is performed using TSMC 90nm Low Power library and commercial memories. Gate-level simulations demonstrated that a trade-off between the complexity of the loop buffer architecture and the power benefits of utilizing it exits. Besides, if the application has loops with small to medium execution time percentage of the total application execution time, the usage of loop buffer architectures in order to bring power benefits to the system should be very carefully evaluated. As case studies, the bio-medical application selected reduces 61% of the energy of the instruction memory organization using a single central loop buffer, whereas the cryptographic application selected reduces 45% using a banked central loop buffer.

Bibliography

[1] R.S. Bajwa, M. Hiraki, H. Kojima, D.J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki. Instruction buffering to reduce power in processors for signal processing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 5(4):417–424, dec 1997.

[2] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Energy and performance improvements in microprocessor design using a loop cache. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 378–383, 1999.

[3] D. Black-Schaffer, J. Balfour, W. Dally, V. Parikh, and JongSoo Park. Hierarchical instruction register organization. In *Computer Architecture Letters*, volume 7, pages 41–44, july-dec. 2008.

[4] Francky Catthoor, P. Raghavan, A. Lambrechts, Murali Jayapala, A. Kritikakou, and J. Absar. *Ultra-Low Energy Domain-Specific Instruction-Set Processors.* Springer Publishing Company, Incorporated, 2010.

[5] Virage Logic Corporation. `http://www.viragelogic.com/`, 2010.

[6] M. Dworkin. Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. In *National Institute of Standards and Technology*, 2004.

[7] Zhiguo Ge, H. B. Lim, and Weng Fai Wong. Memory Hierarchy Hardware-Software Co-design in Embedded Systems. *Computer Science*, pages 1–9, 2004.

[8] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000 (June 13). Circulation Electronic Pages: http://circ.ahajournals.org/cgi/content/full/101/23/e215.

[9] J.I. Gomez, P. Marchal, S. Verdoorlaege, L. Pinuel, and L. Catthoor. Optimizing the memory bandwidth with loop morphing. In *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pages 213–223, sept. 2004.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach.* Denise E. M. Penrose. Morgan Kaufmann, 2007.

[11] K. Inoue, V.G. Moshnyaga, and K. Murakarni. A history-based i-cache for low-energy multimedia applications. In *Low Power Electronics and Design, 2002. ISLPED '02. Proceedings of the 2002 International Symposium on*, pages 148–153, 2002.

[12] Murali Jayapala, Francisco Barat, Pieter Op de Beeck, Francky Catthoor, Geert Deconinck, and Henk Corporaal. A low energy clustered instruction memory hierarchy for long instruction word processors. In *PATMOS '02: Proceedings of the 12th International Workshop on Integrated Circuit Design. Power and Timing Modeling, Optimization and Simulation*, pages 258–267, London, UK, 2002. Springer-Verlag.

[13] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 364–373, 28-31 1990.

[14] J. Kin, Munish Gupta, and W.H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 184–193, 1-3 1997.

[15] Lea Hwang Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pages 267–269, 1999.

[16] NIST. *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology (NIST), November 2001.

[17] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, and D. Verkest. Distributed loop controller architecture for multi-threading in uni-threaded vliw processors. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, 6-10 2006.

[18] Romero Legarreta, I., Addison, P.S., Reed, M.J., Grubb, N., Clegg, G.R. and Robertson, C.E. Continuous wavelet transform modulus maxima analysis of the electrocardiogram: beat characterisation and beat-to-beat measurement. In *Wavelets, Multiresolution and Information Process*, number 1, pages 19–42, 2005.

[19] Weiyu Tang, R. Gupta, and A. Nicolau. Power savings in embedded processors through decode filter cache. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 443–448, 2002.

[20] Target Compiler Technologies. `http://www.retarget.com/`, 2010.

[21] I. Tsekoura. *Design exploration of Application Specific Instruction-Set Cryptographic Processors for resources constrainded systems.* Master thesis, Dept. of Computer Eng. and Informatics, Univ. of Patras and IMEC, 2010.

[22] Manish Verma and Peter Marwedel. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors.* Springer Publishing Company, Incorporated, 2007.

[23] Jason Villarreal, Roman Lysecky, Susan Cotterell, and Frank Vahid. A Study on the Loop Behavior of Embedded Programs. Technical Report UCR-CSE-01-03, University of California, Riverside, December 2001.

[24] K. Vivekanandarajah, T. Srikanthan, and S. Bhattacharyya. Dynamic filter cache for low power instruction memory hierarchy. In *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, pages 607–610, 31 2004.

[25] Cadence Design System Website, 2010.

[26] Synopsys Website, 2010.

[27] Yahya H. Yassin. *Ultra Low Power Application Specific Instruction-Set Processor design for a cardiac beat detector algorithm.* Master thesis, Dept. of Electronics and Telecommunications, Norwegian University of Science and Technology and IMEC, 2009.

[28] Shi Yunhui and Ruan Qiuqi. Continuous wavelet transforms. In *Signal Processing, 2004. Proceedings. ICSP '04. 2004 7th International Conference on*, volume 1, pages 207–210, aug.-4 sept. 2004.

[29] Chuanjun Zhang. An efficient direct mapped instruction cache for application-specific embedded systems. In *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, pages 45–50, sept. 2005.

[30] Hongtao Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for vliw processors. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 197–206, 17-21 2005.

[31] Hongtao Zhong, S.A. Lieberman, and S.A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 25–36, 10-14 2007.